Intel® Open Source HD Graphics and Intel Iris™ Plus Graphics

Programmer's Reference Manual

For the 2016 - 2017 Intel Core™ Processors, Celeron™ Processors, and Pentium™ Processors based on the "Kaby Lake" Platform

Volume 7: 3D-Media-GPGPU

January 2017, Revision 1.0

Creative Commons License

You are free to Share - to copy, distribute, display, and perform the work under the following conditions:

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **No Derivative Works.** You may not alter, transform, or build upon this work.

Notices and Disclaimers

Table of Contents

3D-Compute Front End and Software Interface

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

The registers detailed in this chapter are used across the SNB+ family of products and are extentions 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.

Predication

Predicate Render Registers

Register
MI_PREDICATE_SRC0 - Predicate Rendering Temporary Register0
MI_PREDICATE_SRC1 - Predicate Rendering Temporary Register1
MI_PREDICATE_DATA - Predicate Rendering Data Storage
MI_PREDICATE_RESULT - Predicate Rendering Data Result
MI_PREDICATE_RESULT_1 - Predicate Rendering Data Result 1
MI_PREDICATE_RESULT_2 - Predicate Rendering Data Result 2

MI_SET_PREDICATE

MI_SET_PREDICATE is a command that allows the driver to conditionally execute or skip a command during execution time, as detailed in the instruction definition:

The following is a list of commands that can be programmed when the PREDICATE ENABLE field in MI_SET_PREDICATE allows predication. Commands not listed here will have undefined behavior when executed with predication enabled:

Command
3DSTATE_URB_VS
3DSTATE_URB_HS
3DSTATE_URB_DS
3DSTATE_URB_GS
3DSTATE_PUSH_CONSTANT_ALLOC_VS
3DSTATE_PUSH_CONSTANT_ALLOC_HS
3DSTATE_PUSH_CONSTANT_ALLOC_DS
3DSTATE_PUSH_CONSTANT_ALLOC_GS
3DSTATE_PUSH_CONSTANT_ALLOC_PS
MI_LOAD_REGISTER_IMM
3DSTATE_WM_HZ_OP

Command
MEDIA_VFE_STATE
MEDIA_OBJECT
MEDIA_OBJECT_WALKER
MEDIA_INTERFACE_DESCRIPTOR_LOAD

MI_PREDICATE

The MI_PREDICATE command is used to control the Predicate state bit, which in turn can be used to enable/disable the processing of 3DPRIMITIVE commands.

MI_PREDICATE

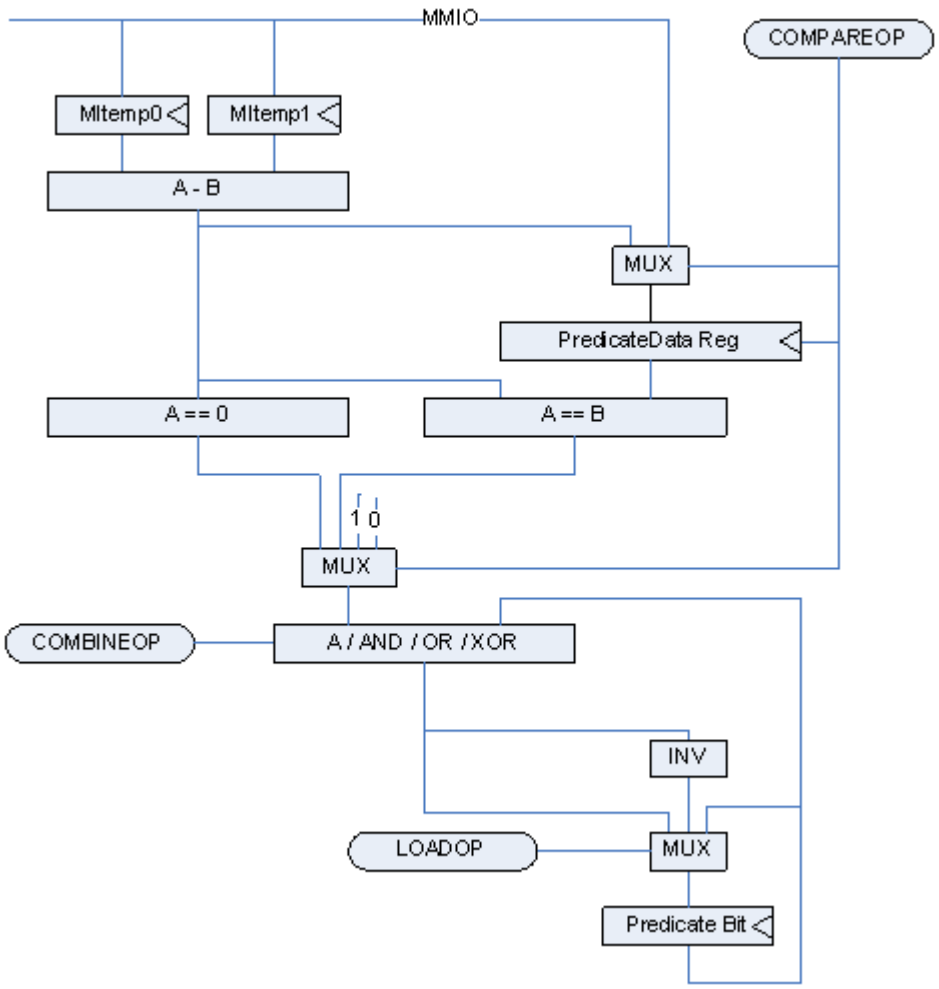
Predicated Rendering Support in HW

DX10 defines predicated rendering, where sequences of rendering commands can be discarded based on the result of a previous predicate test. A new state bit, Predicate, has been added to the command stream. In addition, a PredicateEnable bit is added to 3DPRIMITIVE. When the PredicateEnable bit is set, the command is ignored if the Predicate state bit is set.

A new command, MI_PREDICATE, is added. It contains several control fields which specify how the Predicate bit is generated.

Refer to the diagram below and the command description (linked above) for details.

MI_PREDICATE Function



MI_LOAD_REGISTER_MEM commands can be used to load the MItemp0, MItemp1, and PredicateData registers prior to MI_PREDICATE. To ensure the memory sources of the MI_LOAD_REGISTER_MEM commands are coherent with previous 3D_PIPECONTROL store-DWord operations, software can use the new Pipe Control Flush Enable bit in the PIPE_CONTROL command.

3D-Compute Front End and Software Interface

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

The registers detailed in this chapter are used across the SNB+ family of products and are extentions 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.

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

Registers
GAM_CTX - GAM Context Save Register

Mode and Misc Ctrl Registers

This section contains various registers for controls and modes

Controls/Modes
MI_MODE - Mode Register for Software Interface
GFX_MODE - Graphics Mode Register
GT_MODE - GT Mode Register
SAMPLER_MODE - SAMPLER Mode Register
CACHE_MODE_0 - Cache Mode Register 0
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

B/D/F/Type:

Address Offset: 0x7034

Default Value: 60000060h

Access: RW; RO;

Size: 32 bit

Below Register provides GT2 based L3 sizes.

For GT1 – all sizes need to be multiplied by 0.5.

For GT3 – all sizes need to be multiplied by 2.

For GT4 – all sizes need to be multiplied by 3.

All L3 ways have to be included in the programming to ensure that no ways are left out.

L3CNTLREG - L3 Control 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_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
TIMESTAMP - Reported Timestamp Count
Stream Output Num Primitives Written Counter
Stream Output Write Offsets
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 reun time, save/restored during a GPGPU context switch flow.

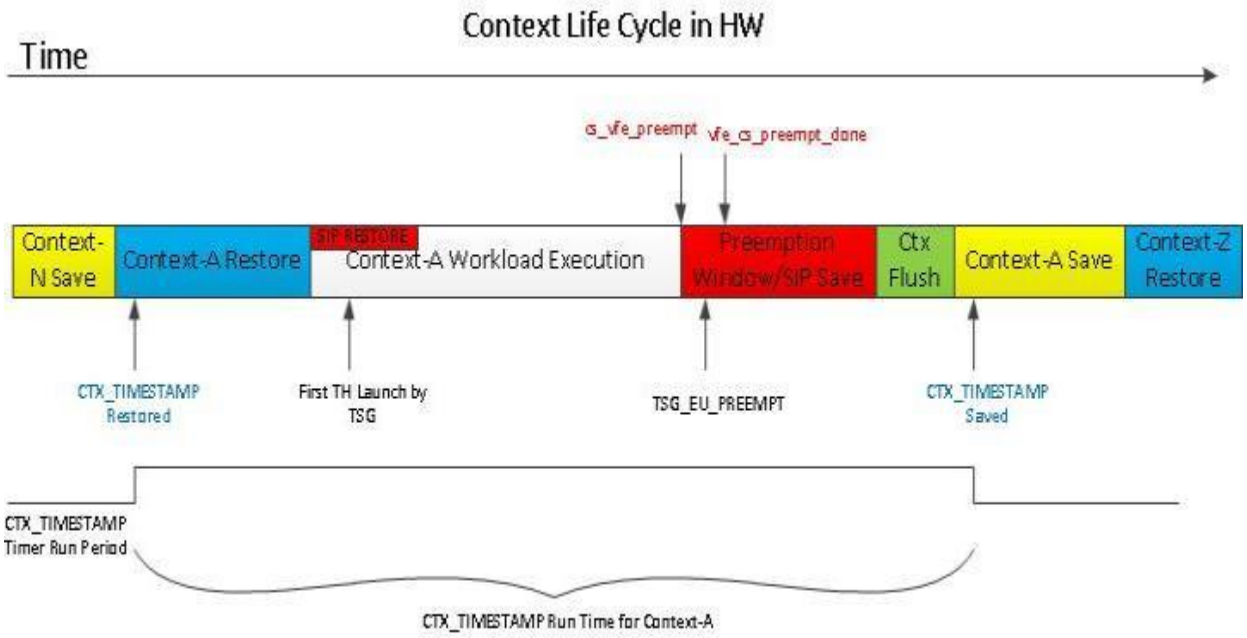


Fig: CTX_TIMESTAMP fucntionality during context execution

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

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

CS ALU Programming and Design

Command streamer implements a rudimentary ALU which supports basic Arithmetic (Addition and Subtraction) and logical operations (AND, OR, XOR) on two 64bit operands. ALU has two 64bit registers at the input SRCA and SRCB to which the operands should be loaded on which operations will be performed and outputted to a 64 bit Accumulator. Zero Flag and Carry Flag are set based on accumulator output.

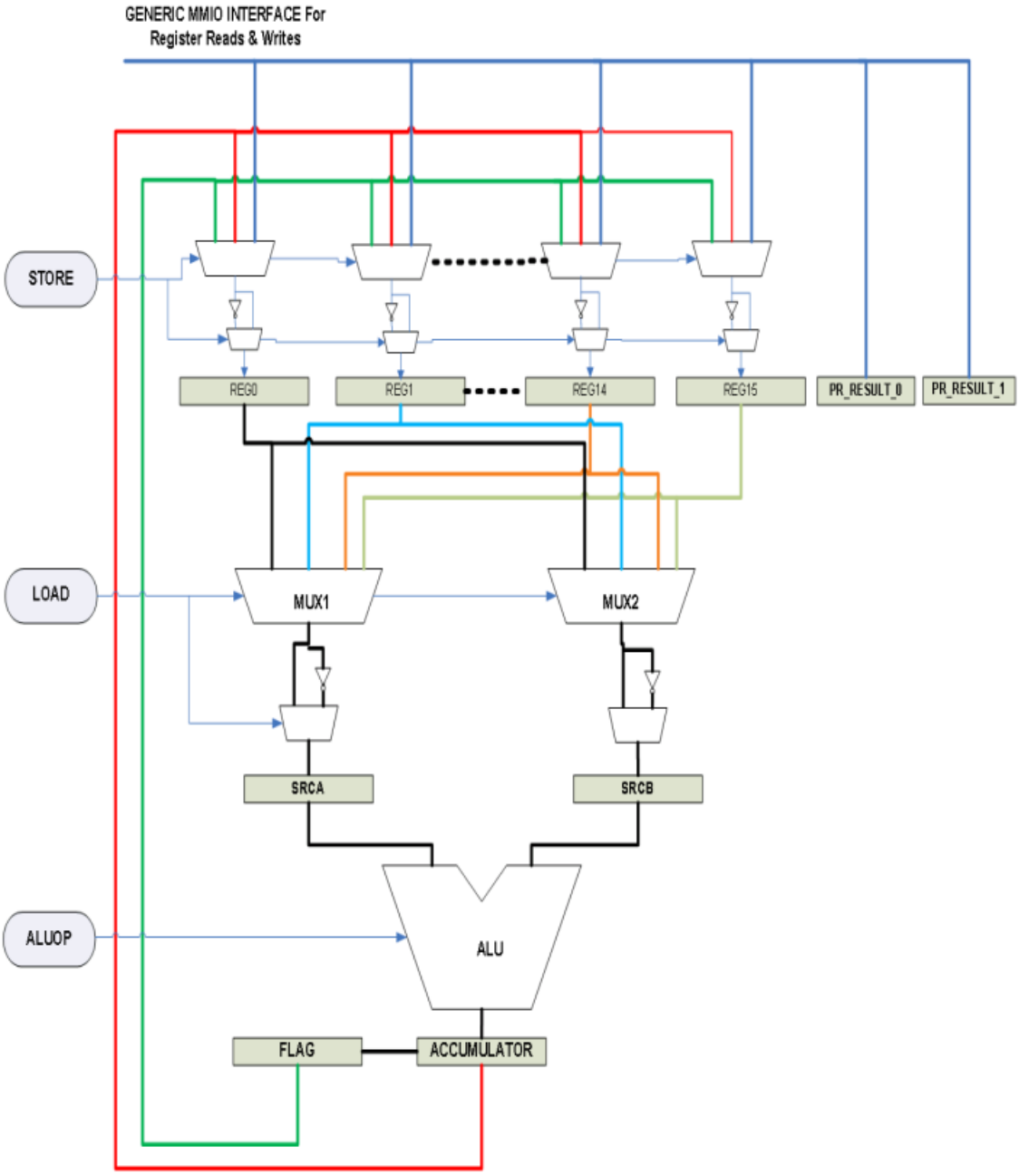
CS ALU Programming and Design

Command streamer implements a rudimentary ALU which supports basic Arithmetic (Addition and Subtraction) and logical operations (AND, OR, XOR) on two 64bit operands. ALU has two 64bit registers at the input SRCA and SRCB to which the operands should be loaded on which operations will be performed and outputted to a 64 bit Accumulator. Zero Flag and Carry Flag are set based on accumulator output.

Generic Purpose Registers

Command streamer implements sixteen 64 bit General Purpose Registers which are MMIO mapped. These registers can be accessed similar to any other MMIO mapped registers through LRI, SRM, LRR, LRM or CPU access path for reads and writes. These registers will be labeled as R0, R1, … R15 throughout the discussion. Refer table in the B-spec update section mapping these registers to corresponding MMIO offset. A selected GPR register can be moved to SRCA or SRCB register using “LOAD” instruction. Outputs of the ALU, Accumulator, ZF and CF can be moved to any of the GPR using “STORE” instruction.

ALU BLOCK Diagram



Instruction Set

The instructions supported by the ALU can be broadly categorized into three groups:

- To move data from GPR to SRCA/SRCB – LOAD instruction.
- To move data from ACCUMULATOR/CF/ZF to GPR – STORE Instruction.

- To do arithmetic/Logical operations on SRCA and SRCB of ALU - ADD/SUB/AND/XOR/OR. Note: Accumulator is loaded with value of SRCA - SRCB on a subtraction.

Instruction Format

Each instruction is one Dword in size and consists of an ALU OPCODE, OPERAND1 and OPERAND2 in the format shown below.

ALU OPCODE	Operand-1	Operand-2
12 bits	10 bits	10 bits

LOAD Operation

The LOAD instruction moves the content of the destination register (Operand2) into the source register (Operand1). The destination register can be any of the GPR (R0, R1, ..., R15) and the source registers are SRCA and SRCB of the ALU. This is the only means SRCA and SRCB can be programmed.

LOAD has different flavors, wherein one can load the inverted version of the source register into the destination register or a hard coded value of all Zeros and All ones.

```
 // Loads any of Reg0 to Reg15 into the SRCA or SRCB registers of ALU.

     LOAD <SRCA, SRCB>, <REG0..REG15>

 // Loads inverted (bit wise) value of the mentioned Reg0 to 15 into SRCA or SRCB registers of
ALU.

     LOADINV <SRCA, SRCB>, <REG0..REG15>

 // Loads "0" into SRCA or SRCB

     LOAD0 <SRCA, SRCB>

 // Loads "1" into SRCA or SRCB

     LOAD1 <SRCA, SRCB>
```

31	20	19	10	9	0
Opcode		**Operand1**		**Operand2**	
LOAD		SRCA/SRCB		R0,R1..R15	
LOADINV		SRCA/SRCB		R0,R1..R15	
LOAD0		SRCA/SRCB		N/A	
LOAD1		SRCA/SRCB		N/A	

Arithmetic/Logical Operations

ADD, SUB, AND, OR, and XOR are the Arithmetic and Logical operations supported by Arithmetic Logic Unit (ALU). When opcode corresponding to a logical operation is performed on SRCA and SRCB, the

result is sent to ACCUMULATOR (ACCU), CF and ZF. Note that ACCU is 64-bit register. A NOOP when submitted to the ALU doesn't do anything, it is meant for creating bubble or kill cycles.

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
ADD	N/A	N/A
SUB	N/A	N/A
AND	N/A	N/A
OR	N/A	N/A
XOR	N/A	N/A
NOOP	N/A	NA

STORE Operation

The STORE instruction moves the content of the destination register (Operand1) into the source register (Operand2). The destination register can be accumulator (ACCU), CF or ZF. STORE has different flavors, wherein one can load the inverted version of the source register into destination register via STOREINV. When CF or ZF are stored, the same value is replicated on all 64 bits.

```
// Loads ACCMULATOR or Carry Flag or Zero Flag in to any of the generic registers
// Reg0 to Reg16. In case of CF and ZF same value is replicated on all the 64 bits.

    STORE   <R0.. R15>, <ACCU, CF, ZF >

// Loads inverted (ACCMULATOR or Carry Flag or Zero Flag) in to any of the
// generic registers Reg0 to Reg15.

    STOREINV <R0.. R15>, <ACCU, CF, ZF>
```

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
STORE	R0,R1..R15	ACCU/ZF/CF
STOREINV	R0, R1.. R15	ACCU/ZF/CF

Summary for ALU

Total Opcodes Supported: 12

Total Addressable Registers as source or destination: 21

- 16 GPR (R0, R1 …R15)
- 1 ACCU
- 1ZF
- 1CF
- SRCA, SRCB

Summary of Instructions Supported

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
LOAD	SRCA/SRCB	REG0..REG15
LOADINV	SRCA/SRCB	REG0..REG15
LOAD0	SRCA/SRCB	N/A
LOAD1	SRCA/SRCB	N?A
ADD	N/A	N/A
SUB	N/A	N/A
AND	N/A	N/A
OR	N/A	N/A
XOR	N/A	N/A
NOOP	N/A	N/A
STORE	REG0..REG15	ACCU/CF/ZF
STOREINV	REG0..REG15	ACCU/CF/ZF

Table for ALU OPCODE Encodings

ALU OPCODE	OPCODE ENCODING
NOOP	0x000
LOAD	0x080
LOADINV	0x480
LOAD0	0x081
LOAD1	0x481
ADD	0x100
SUB	0x101
AND	0x102
OR	0x103
XOR	0x104
STORE	0x180
STOREINV	0x580

In the above mentioned table, ALU Opcode Encodings look like random numbers. The rationale behind those encodings is because the ALU Opcode is further broken down into sub-sections for ease-of-design implementation.

PREFIX		OPCODE		SUBOPCODE	
11	10	9	7	6	0
PREFIX VALUE		**Description**			
0		Regular			
1		Invert			
OPCODE VALUE		**Description**			
0		NOOP			
1		LOAD			
2		ALU			
3		STORE			

ALU OPCODE	ENCODING	PREFIX	OPCODE	SUBOPCODE
		10	9 7	6 0
NOOP	0x000	0	0	0
LOAD	0x080	0	1	0
LOADINV	0x480	1	1	0
LOAD0	0x081	0	1	1
LOAD1	0x481	1	1	1
ADD	0x100	0	2	0
SUB	0x101	0	2	1
AND	0x102	0	2	2
OR	0x103	0	2	3
XOR	0x104	0	2	4
STORE	0x180	0	3	0
STOREINV	0x580	1	3	0

Table for Register Encodings

Register	Register Encoding
R0	0x0
R1	0x1
R2	0x2
R3	0x3
R4	0x4
R5	0x5
R6	0x6
R7	0x7

Register	Register Encoding
R8	0x8
R9	0x9
R10	0xa
R11	0xb
R12	0xc
R13	0xd
R14	0xe
R15	0xf
SRCA	0x20
SRCB	0x21
ACCU	0x31
ZF	0x32
CF	0x33

CS_GPR - Command Streamer General Purpose Registers

Following are Command Streamer General Purpose Registers:

CS_GPR - General Purpose Register

Memory Interface Commands for Rendering Engine

Command
MI_SET_CONTEXT
MI_TOPOLOGY_FILTER
MI_URB_ATOMIC_ALLOC
MI_LOAD_URB_MEM
MI_STORE_URB_MEM

State Commands

This section covers the following commands:

- **STATE_PREFETCH command.** The STATE_PREFETCH command is provided strictly as an optional mechanism to possibly enhance pipeline performance by prefetching data into the GPE's Instruction and State Cache (ISC).
- **STATE_SIP command**

Command
STATE_SIP
3DSTATE_URB_CLEAR

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.

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 (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 restore of these packets. If the pipe control invalidate operation is completed before the context save, the indirect pointers will not be restored from memory.

- Pipeline State Pointer
- Media State Pointer
- Constant Buffer Packet

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

Description
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.
Protected Mem Enable	22	Requires stall bit ([20] of DW1) set.
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.
Indirect State Pointers Disable	9	Requires stall bit ([20] of DW1) set.
Store Data Index	21	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'.
Sync GFDT	17	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0' or 0x2520[13] must be set.
TLB inv	18	Requires stall bit ([20] of DW1) set.
Post Sync Op	15:14	LRI Post Sync Operation ([23] of DW1) must be set to '0'.
Protected Memory Application ID	6	Requires stall bit ([20] of DW1) set.

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
Stall Pixel Scoreboard	1	No Restriction.
Inst invalidate	11	No Restriction.
Tex invalidate	10	Requires stall bit ([20] of DW) set for all GPGPU Workloads.
VF invalidate	4	“Post Sync Operation” must be enabled to “Write Immediate Data” or “Write PS Depth Count” or “Write Timestamp”.
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.

Context Image

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

Logical Contexts are aligned to 256-byte boundaries.

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

Power Context Image

Render Engine Power Context

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.

RCS Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		CS	1	0	CSFE
Load_Register_Immediate header	0x1100_108B	CS	1	0001	CSFE
GFX_MODE	0x229C	CS	2	0002	CSFE
GHWSP	0x2080	CS	2	0004	CSFE
RING_BUFFER_CONTROL (Ring Always Disabled)	0x203C	CS	2	0006	CSFE
Ring Head Pointer Register	0x2034	CS	2	0008	CSFE
Ring Tail Pointer Register	0x2030	CS	2	000A	CSFE
RING_BUFFER_START	0x2038	CS	2	000C	CSFE
RING_BUFFER_CONTROL (Original status)	0x203C	CS	2	000E	CSFE
Batch Buffer Current Head Register (UDW)	0x2168	CS	2	0010	CSFE
Batch Buffer Current Head Register	0x2140	CS	2	0012	CSFE
Batch Buffer State Register	0x2110	CS	2	0014	CSFE
SECOND_BB_ADDR_UDW	0x211C	CS	2	0016	CSFE
SECOND_BB_ADDR	0x2114	CS	2	0018	CSFE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
SECOND_BB_STATE	0x2118	CS	2	001A	CSFE
RC_PSMI_CONTROL	0x2050	CS	2	001C	CSFE
RC_PWRCTX_MAXCNT	0x2054	CS	2	001E	CSFE
CTX_WA_PTR	0x2058	CS	2	0020	CSFE
NOPID	0x2094	CS	2	0022	CSFE
HWSTAM	0x2098	CS	2	0024	CSFE
IMR	0x20A8	CS	2	0026	CSFE
EIR	0x20B0	CS	2	0028	CSFE
EMR	0x20B4	CS	2	002A	CSFE
CMD_CCTL_0	0x20C4	CS	2	002C	CSFE
UHPTR	0x2134	CS	2	002E	CSFE
BB_PREEMPT_ADDR_UDW	0x216C	CS	2	0030	CSFE
BB_PREEMPT_ADDR	0x2148	CS	2	0032	CSFE
RING_BUFFER_HEAD_PREEMPT_REG	0x214C	CS	2	0034	CSFE
PREEMPT_DLY	0x2214	CS	2	0036	CSFE
CTXT_PREMP_DBG	0x2248	CS	2	0038	CSFE
SYNC_FLIP_STATUS	0x22D0	CS	2	003A	CSFE
SYNC_FLIP_STATUS_1	0x22D4	CS	2	003C	CSFE
SYNC_FLIP_STATUS_2	0x22EC	CS	2	003E	CSFE
WAIT_FOR_RC6_EXIT	0x20CC	CS	2	0040	CSFE
RCS_CTXID_PREEMPTION_HINT	0x24CC	CS	2	0042	CSFE
CS_PREEMPTION_HINT_UDW	0x24C8	CS	2	0044	CSFE
CS_PREEMPTION_HINT	0x24BC	CS	2	0046	CSFE
CCID Register	0x2180	CS	2	0048	CSFE
SBB_PREEMPT_ADDRESS_UDW	0x2138	CS	2	004A	CSFE
SBB_PREEMPT_ADDRESS	0x213C	CS	2	004C	CSFE
MI_PREDICATE_RESULT_2	0x23BC	CS	2	004E	CSFE
CTXT_ST_PTR	0x23A0	CS	2	0050	CSFE
CTXT_ST_BUF	0x2370	CS	24	0052	CSFE
SEMA_WAIT_POLL	0x224C	CS	2	006A	CSFE
IDLEDELAY	0x223C	CS	2	006C	CSFE
DISPLAY MESSAGE FORWARD STATUS	0x22E8	CS	2	006E	CSFE
RCS_FORCE_TO_NONPRIV	0x24D0	CS	24	0070	CSFE
EXECLIST_STATUS_REGISTER	0x2234	CS	2	0088	CSFE
CXT_OFFSET	0x21AC	CS	2	008C	CSBE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		CS	2	008E	CSFE
NOOP		CS	1	0090	CSBE
Load_Register_Immediate header	0x1100_1021	CS	1	0091	CSBE
FF_THREAD_MODE	0x20A0	CS	2	0096	CSBE
GAFS_Mode	0x212C	CS	2	009C	CSBE
CXT_SIZE	0x21A8	CS	2	009E	CSBE
RS_OFFSET	0x21B4	CS	2	00A4	CSBE
RS_PREEMPTION_HINT_UDW	0x24C4	CS	2	00A6	CSBE
RS_PREEMPTION_HINT	0x24C0	CS	2	00A8	CSBE
URB_CTX_OFFSET	0x21B8	CS	2	00AA	CSBE
VF_CTX_OFFSET	0x21BC	CS	2	00AC	CSBE
VF PREMPTION VERTEX HINT	0x83B0	VF	2	00B0	CSBE
VF PREEMPTION INSTANCE HINT	0x83B4	VF	2	00B2	CSBE
NOOP		CS	10	00B4	CSBE
NOOP		CS	1	00BE	CSBE
MI_BATCH_BUFFER_END		CS	1	00BF	CSBE

Engine Register and State Context

This section describes programming requirements for the Register State Context.

Programming Note	
Context:	Register State Context

- All the MMIO registers part of the “Engine Register and State Context Image” are context specific and gets context save/restored upon a context switch. MMIO register values belonging to a context can be exercised through HOST/IA MMIO interface only when the context is active in HW. Exercising context specific MMIO registers through HOST/IA MMIO is completely asynchronous to the context execution in HW and can’t guarantee a desired sampling point during execution. In execlist mode of scheduling there is no active context when HW is Idle.
- All the write access to MMIO registers listed in the “Engine Register and State Context image” subsections below must be done through MI commands (MI_LOAD_REGISTER_IMM, MI_LOAD_REG_MEM, MI_LOAD_REGISTER_REG) in the command sequence.

LRCA Context Image

EXECLIST CONTEXT(Ring)
EXECLIST CONTEXT (PPGTT Base)
ENGINE CONTEXT(CS)
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(SOL)
ENGINE CONTEXT(VF)
ENGINE CONTEXT(VFE)
EXTENDED ENGINE CONTEXT(RS)
URB_ATOMIC CONTEXT(URB)

Register State Context

Context Color Codes Used

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

Register Information

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
NOOP		CS	1	0
Load_Register_Immediate header	0x1100_101B	CS	1	0001
Context Control	0x2244	CS	2	0002
Ring Head Pointer Register	0x2034	CS	2	0004
Ring Tail Pointer Register	0x2030	CS	2	0006
RING_BUFFER_START	0x2038	CS	2	0008
RING_BUFFER_CONTROL	0x203C	CS	2	000A
Batch Buffer Current Head Register (UDW)	0x2168	CS	2	000C
Batch Buffer Current Head Register	0x2140	CS	2	000E
Batch Buffer State Register	0x2110	CS	2	0010
SECOND_BB_ADDR_UDW	0x211C	CS	2	0012
SECOND_BB_ADDR	0x2114	CS	2	0014
SECOND_BB_STATE	0x2118	CS	2	0016
BB_PER_CTX_PTR	0x21C0	CS	2	0018
RCS_INDIRECT_CTX	0x21C4	CS	2	**001A**
RCS_INDIRECT_CTX_OFFSET	0x21C8	CS	2	**001C**
NOOP		CS	2	001E
NOOP		CS	1	0020
Load_Register_Immediate header	0x1100_1011	CS	1	0021
CTX_TIMESTAMP	0x23A8	CS	2	0022
PDP3_UDW0x1100_1029	0x228C	CS	2	0024
PDP3_LDW	0x2288	CS	2	0026
PDP2_UDW	0x2284	CS	2	0028
PDP2_LDW	0x2280	CS	2	002A
PDP1_UDW	0x227C	CS	2	002C
PDP1_LDW	0x2278	CS	2	002E
PDP0_UDW	0x2274	CS	2	0030
PDP0_LDW	0x2270	CS	2	0032
NOOP		CS	12	0034
NOOP		CS	1	0040
Load_Register_Immediate header	0x1100_0001	CS	1	0041
R_PWR_CLK_STATE	0x20C8	CS	2	0042
GPGPU_CSR_BASE_ADDRESS		CS	3	0044

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
NOOP		CS	9	0047
NOOP		CS	1	0050
Load_Register_Immediate header	0x1100_1057	CS	1	0051
EXCC	0x2028	CS	2	0052
MI_MODE	0x209C	CS	2	0054
INSTPM	0x20C0	CS	2	0056
PR_CTR_CTL	0x2178	CS	2	0058
PR_CTR_THRSH	0x217C	CS	2	005A
TIMESTAMP Register (LSB)	0x2358	CS	2	005C
BB_START_ADDR_UDW	0x2170	CS	2	005E
BB_START_ADDR	0x2150	CS	2	0060
BB_ADD_DIFF	0x2154	CS	2	0062
BB_OFFSET	0x2158	CS	2	0064
MI_PREDICATE_RESULT_1	0x241C	CS	2	0066
CS_GPR (1-16)	0x2600	CS	64	0068
IPEHR	0x2068	CS	2	**00A8**
NOOP		CS	6	**00AA**
NOOP		CS	1	00B0
Load_Register_Immediate header	0x1100_1085	CS	1	00B1
CS_CONTEXT_STATUS1	0x2184	CS	2	00B4
IA_VERTICES_COUNT	0x2310	CS	4	00B6
IA_PRIMITIVES_COUNT	0x2318	CS	4	00BA
VS_INVOCATION_COUNT	0x2320	CS	4	00BE
HS_INVOCATION_COUNT	0x2300	CS	4	00C2
DS_INVOCATION_COUNT	0x2308	CS	4	00C6
GS_INVOCATION_COUNT	0x2328	CS	4	00CA
GS_PRIMITIVES_COUNT	0x2330	CS	4	00CE
CL_INVOCATION_COUNT	0x2338	CS	4	00D2
CL_PRIMITIVES_COUNT	0x2340	CS	4	00D6
PS_INVOCATION_COUNT_0	0x22C8	CS	4	00DA
PS_DEPTH_COUNT _0	0x22D8	CS	4	00DE
GPUGPU_DISPATCHDIMX	0x2500	CS	2	00E4
GPUGPU_DISPATCHDIMY	0x2504	CS	2	00E6
GPUGPU_DISPATCHDIMZ	0x2508	CS	2	00E8

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
MI_PREDICATE_SRC0	0x2400	CS	2	00EA
MI_PREDICATE_SRC0	0x2404	CS	2	00EC
MI_PREDICATE_SRC1	0x2408	CS	2	00EE
MI_PREDICATE_SRC1	0x240C	CS	2	00F0
MI_PREDICATE_DATA	0x2410	CS	2	00F2
MI_PREDICATE_DATA	0x2414	CS	2	00F4
MI_PRED_RESULT	0x2418	CS	2	00F6
3DPRIM_END_OFFSET	0x2420	CS	2	00F8
3DPRIM_START_VERTEX	0x2430	CS	2	00FA
3DPRIM_VERTEX_COUNT	0x2434	CS	2	00FC
3DPRIM_INSTANCE_COUNT	0x2438	CS	2	00FE
3DPRIM_START_INSTANCE	0x243C	CS	2	0100
3DPRIM_BASE_VERTEX	0x2440	CS	2	0102
GPGPU_THREADS_DISPATCHED	0x2290	CS	4	0104
PS_INVOCATION_COUNT_1	0x22F0	CS	4	0108
PS_DEPTH_COUNT _1	0x22F8	CS	4	010C
RS_PREEMPT_STATUS	0x215C	CS	2	0110
CTX_SEMA_REG	0x24B4	CS	4	0112
PRODUCE_COUNT_BTP	0x2480	CS	2	0116
PRODUCE_COUNT_DX9_CONSTANTS	0x2484	CS	2	0118
PRODUCE_COUNT_GATHER_CONSTANTS	0x248C	CS	2	011A
PARSED_COUNT_BTP	0x2490	CS	2	011C
PARSED_COUNT_DX9_CONSTANTS	0x2494	CS	2	011E
OA_CTX_CONTROL	0x2360	CS	2	0128
OACTXID	0x2364	CS	2	012A
PS_INVOCATION_COUNT_2	0x2448	CS	4	012C
PS_DEPTH_COUNT_2	0x2450	CS	4	0130
RS_PREEMPT_STATUS_UDW	0x2174	CS	2	0134
NOOP		CS	8	0138
MI_TOPOLOGY_FILTER		CS	1	0140
NOOP		CS	2	0141
PIPELINE_SELECT		CS	1	0143
STATE_BASE_ADDRESS		CS	19	0144
3DSTATE_PUSH_CONSTANT_ALLOC_VS		CS	2	0157

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
3DSTATE_PUSH_CONSTANT_ALLOC_HS		CS	2	0159
3DSTATE_PUSH_CONSTANT_ALLOC_DS		CS	2	015B
3DSTATE_PUSH_CONSTANT_ALLOC_GS		CS	2	015D
3DSTATE_PUSH_CONSTANT_ALLOC_PS		CS	2	015F
3DSTATE_BINDING_TABLE_POOL_ALLOC		CS	4	0161
3DSTATE_GATHER_POOL_ALLOC		CS	4	0165
3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC		CS	4	0169
MI_RS_CONTROL		CS	1	016D
MI_URB_ATOMIC_ALLOC		CS	1	016E
NOOP		CS	17	016F
NOOP		GAMT	1	0180
Load_Register_Immediate header	0x1100_108B	GAMT	1	0181
GFX_MOCS_0	C800	GAMT	2	0182
GFX_MOCS_1	C804	GAMT	2	0184
GFX_MOCS_2	C808	GAMT	2	0186
GFX_MOCS_3	C80C	GAMT	2	0188
GFX_MOCS_4	C810	GAMT	2	018A
GFX_MOCS_5	C814	GAMT	2	018C
GFX_MOCS_6	C818	GAMT	2	018E
GFX_MOCS_7	C81C	GAMT	2	0190
GFX_MOCS_8	C820	GAMT	2	0192
GFX_MOCS_9	C824	GAMT	2	0194
GFX_MOCS_10	C828	GAMT	2	0196
GFX_MOCS_11	C82C	GAMT	2	0198
GFX_MOCS_12	C830	GAMT	2	019A
GFX_MOCS_13	C834	GAMT	2	019C
GFX_MOCS_14	C838	GAMT	2	019E
GFX_MOCS_15	C83C	GAMT	2	01A0
GFX_MOCS_16	C840	GAMT	2	01A2
GFX_MOCS_17	C844	GAMT	2	01A4
GFX_MOCS_18	C848	GAMT	2	01A6
GFX_MOCS_19	C84C	GAMT	2	01A8
GFX_MOCS_20	C850	GAMT	2	01AA
GFX_MOCS_21	C854	GAMT	2	01AC

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
GFX_MOCS_22	C858	GAMT	2	01AE
GFX_MOCS_23	C85C	GAMT	2	01B0
GFX_MOCS_24	C860	GAMT	2	01B2
GFX_MOCS_25	C864	GAMT	2	01B4
GFX_MOCS_26	C868	GAMT	2	01B6
GFX_MOCS_27	C86C	GAMT	2	01B8
GFX_MOCS_28	C870	GAMT	2	01BA
GFX_MOCS_29	C874	GAMT	2	01BC
GFX_MOCS_30	C878	GAMT	2	01BE
GFX_MOCS_31	C87C	GAMT	2	01C0
GFX_MOCS_32	C880	GAMT	2	01C2
GFX_MOCS_33	C884	GAMT	2	01C4
GFX_MOCS_34	C888	GAMT	2	01C6
GFX_MOCS_35	C88C	GAMT	2	01C8
GFX_MOCS_36	C890	GAMT	2	01CA
GFX_MOCS_37	C894	GAMT	2	01CC
GFX_MOCS_38	C898	GAMT	2	01CE
GFX_MOCS_39	C89C	GAMT	2	01D0
GFX_MOCS_40	C8A0	GAMT	2	01D2
GFX_MOCS_41	C8A4	GAMT	2	01D4
GFX_MOCS_42	C8A8	GAMT	2	01D6
GFX_MOCS_43	C8AC	GAMT	2	01D8
GFX_MOCS_44	C8B0	GAMT	2	01DA
GFX_MOCS_45	C8B4	GAMT	2	01DC
GFX_MOCS_46	C8B8	GAMT	2	01DE
GFX_MOCS_47	C8BC	GAMT	2	01E0
GFX_MOCS_48	C8C0	GAMT	2	01E2
GFX_MOCS_49	C8C4	GAMT	2	01E4
GFX_MOCS_50	C8C8	GAMT	2	01E6
GFX_MOCS_51	C8CC	GAMT	2	01E8
GFX_MOCS_52	C8D0	GAMT	2	01EA
GFX_MOCS_53	C8D4	GAMT	2	01EC
GFX_MOCS_54	C8D8	GAMT	2	01EE
GFX_MOCS_55	C8DC	GAMT	2	01F0

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
GFX_MOCS_56	C8E0	GAMT	2	01F2
GFX_MOCS_57	C8E4	GAMT	2	01F4
GFX_MOCS_58	C8E8	GAMT	2	01F6
GFX_MOCS_59	C8EC	GAMT	2	01F8
GFX_MOCS_60	C8F0	GAMT	2	01FA
GFX_MOCS_61	C8F4	GAMT	2	01FC
GFX_MOCS_62	C8F8	GAMT	2	01FE
GFX_MOCS_63	C8FC	GAMT	2	0200
Tiled Resources VA Translation Table L3 Pointer	4DE0	GAMT	4	0202
Tiled Resources Null Tile Detection Register	4DE8	GAMT	2	0206
Tiled Resources Invalid Tile Detection Register	4DEC	GAMT	2	0208
Tiled Resources Virtual Address Detection Registers	4DF0	GAMT	2	020A
Tiled Resources Translation Table Control Register	4DF4	GAMT	2	020C
NOOP		GAM	2	020E
NOOP		LNCF	1	0210
Load_Register_Immediate header	0x1100_1001	LNCF	1	0211
L3CNTLREG	7034	LNCF	2	0212
NOOP		LNCF	1	0214
Load_Register_Immediate header	0x1100_103F	LNCF	1	0215
LNCFCMOCS0	B020	LNCF	2	0216
LNCFCMOCS1	B024	LNCF	2	0218
LNCFCMOCS2	B028	LNCF	2	021A
LNCFCMOCS3	B02C	LNCF	2	021C
LNCFCMOCS4	B030	LNCF	2	021E
LNCFCMOCS5	B034	LNCF	2	0220
LNCFCMOCS6	B038	LNCF	2	0222
LNCFCMOCS7	B03C	LNCF	2	0224
LNCFCMOCS8	B040	LNCF	2	0226
LNCFCMOCS9	B044	LNCF	2	0228
LNCFCMOCS10	B048	LNCF	2	022A
LNCFCMOCS11	B04C	LNCF	2	022C
LNCFCMOCS12	B050	LNCF	2	022E
LNCFCMOCS13	B054	LNCF	2	0230

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
LNCFCMOCS14	B058	LNCF	2	0232
LNCFCMOCS15	B05C	LNCF	2	0234
LNCFCMOCS16	B060	LNCF	2	0236
LNCFCMOCS17	B064	LNCF	2	0238
LNCFCMOCS18	B068	LNCF	2	023A
LNCFCMOCS19	B06C	LNCF	2	023C
LNCFCMOCS20	B070	LNCF	2	023E
LNCFCMOCS21	B074	LNCF	2	0240
LNCFCMOCS22	B078	LNCF	2	0242
LNCFCMOCS23	B07C	LNCF	2	0244
LNCFCMOCS24	B080	LNCF	2	0246
LNCFCMOCS25	B084	LNCF	2	0248
LNCFCMOCS26	B088	LNCF	2	024A
LNCFCMOCS27	B08C	LNCF	2	024C
LNCFCMOCS28	B090	LNCF	2	024E
LNCFCMOCS29	B094	LNCF	2	0250
LNCFCMOCS30	B098	LNCF	2	0252
LNCFCMOCS31	B09C	LNCF	2	0254
NOOP		LNCF	10	0256
3DSTATE_CONSTANT_VS_Commited		SVG	11	0260
3DSTATE_CONSTANT_HS_Commited		SVG	11	026B
3DSTATE_CONSTANT_DS_Commited		SVG	11	0276
3DSTATE_CONSTANT_GS_Commited		SVG	11	0281
3DSTATE_VS		SVG	9	028C
3DSTATE_BINDING_TABLE_POINTERS_VS		SVG	2	0295
3DSTATE_SAMPLER_STATE_POINTERS_VS		SVG	2	0297
3DSTATE_URB_VS		SVG	2	0299
3DSTATE_HS		SVG	9	029B
3DSTATE_BINDING_TABLE_POINTERS_HS		SVG	2	02A4
3DSTATE_SAMPLER_STATE_POINTERS_HS		SVG	2	02A6
3DSTATE_URB_HS		SVG	2	02A8
3DSTATE_TE		SVG	4	02AA
3DSTATE_DS		SVG	11	02AE
3DSTATE_BINDING_TABLE_POINTERS_DS		SVG	2	02B9

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
3DSTATE_SAMPLER_STATE_POINTERS_DS		SVG	2	02BB
3DSTATE_URB_DS		SVG	2	02BD
3DSTATE_GS		SVG	10	02BF
3DSTATE_BINDING_TABLE_POINTERS_GS		SVG	2	02C9
3DSTATE_SAMPLER_STATE_POINTERS_GS		SVG	2	02CB
3DSTATE_URB_GS		SVG	2	02CD
3DSTATE_STREAMOUT		SVG	5	02CF
3DSTATE_CLIP		SVG	4	02D4
3DSTATE_SF		SVG	4	02D8
3DSTATE_SCISSOR_STATE_POINTERS		SVG	2	02DC
3DSTATE_VIEWPORT_STATE_POINTERS_CL_SF		SVG	2	02DE
3DSTATE_RASTER		SVG	5	02E0
3DSTATE_WM_HZ_OP		SVG	5	02E5
3DSTATE_MULTISAMPLE		SVG	2	02EA
3DSTATE_CONSTANT_VS_NonComitted		SVG	11	02EC
3DSTATE_CONSTANT_HS_NonComitted		SVG	11	02F7
3DSTATE_CONSTANT_DS_NonComitted		SVG	11	0302
3DSTATE_CONSTANT_GS_NonComitted		SVG	11	030D
3DSTATE_DRAW_RECTANGULAR		SVG	4	0318
Load_Register_Immediate header	0x1100_1001	SVG	1	031C
FF_PERF_REG	0x6b1c	SVG	2	031D
NOOP		SVG	1	031F
3DSTATE_CONSTANT_PS_comitted		SVL	11	0320
NOOP		SVL	1	032B
3DSTATE_WM		SVL	2	032C
3DSTATE_VIEWPORT_STATE_POINTER_CC		SVL	2	032E
3DSTATE_CC_STATE_POINTERS		SVL	2	0330
3DSATE_WM_SAMPLEMASK		SVL	2	0332
3DSTATE_WM_DEPTH_STENCIL		SVL	4	0334
3DSTATE_WM_CHROMAKEY		SVL	2	0338
3DSTATE_DEPTH_BUFF		SVL	8	033A
3DSTATE_HIZ_DEPTH_BUFF		SVL	5	0342
3DSTATE_STC_DEPTH_BUFF		SVL	5	0347
3DSTATE_CLEAR_PARAMS		SVL	3	034C

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
3DSTATE_SBE		SVL	6	034F
3DSTATE_SBE_SWIZ		SVL	11	0355
3DSTATE_PS		SVL	12	0360
3DSTATE_BINDING_TABLE_POINTERS_PS		SVL	2	036C
STATE_SAMPLER_STATE_POINTERS_PS		SVL	2	036E
3DSTATE_BLEND_STATE_POINTERS		SVL	2	0370
3DSTATE_PS_EXTRA		SVL	2	0372
3DSTATE_PS_BLEND		SVL	2	0374
3DSTATE_CONSTANT_PS_NonComitted		SVL	11	0376
STATE_SIP		SVL	3	0381
3DSTATE_SAMPLE_PATTERN		SVL	9	0384
Load_Register_Immediate header	0x1100_101B	SVL	1	038D
Cache_Mode_0	0x7000	SVL	2	038E
Cache_Mode_1	0x7004	SVL	2	0390
GT_MODE	0x7008	SVL	2	0392
FBC_RT_BASE_ADDR_REGISTER	0x7020	SVL	2	0398
FBC_RT_BASE_ADDR_REGISTER_UPPER	0x7024	SVL	2	039A
SAMPLER_MODE	0x7028	SVL	2	039C
OA_CULL	0x7030	SVL	2	03A2
				03AA
NOOP		SVL	6	**03AA**
NOOP		TDL	1	03B0
Load_Register_Immediate header	0x1100_103B	TDL	1	03B1
TD_CTL	E400	TDL	2	03B2
TD_CTL2	E404	TDL	2	03B4
TD_VF_VS_EMSK	E408	TDL	2	03B6
TD_GS_EMSK	E40C	TDL	2	03B8
TD_WIZ_EMSK	E410	TDL	2	03BA
TD_TS_EMSK	E428	TDL	2	03BC
TD_HS_EMSK	E4B0	TDL	2	03BE
TD_DS_EMSK	E4B4	TDL	2	03C0
EU_PERF_CNT_CTL0	E458	TDL	2	03DE
EU_PERF_CNT_CTL1	E558	TDL	2	03E0
EU_PERF_CNT_CTL2	E658	TDL	2	03E2

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
EU_PERF_CNT_CTL3	E758	TDL	2	03E4
EU_PERF_CNT_CTL4	E45C	TDL	2	03E6
EU_PERF_CNT_CTL5	E55C	TDL	2	03E8
EU_PERF_CNT_CTL6	E65C	TDL	2	03EA
NOOP		TDL	2	03EE
NOOP		WM	1	03F0
Load_Register_Immediate header	0x1100_1007	WM	1	03F1
WMHWCLRVAL	0x5524	WM	2	03F6
3DSTATE_POLY_STIPPLE_PATTERN		WM	33	03FA
3DSTATE_AA_LINE_PARAMS		WM	3	041B
3DSTATE_POLY_STIPPLE_OFFSET		WM	2	041E
3DSTATE_LINE_STIPPLE		WM	3	0420
NOOP		WM	13	0423
NOOP		SC	1	0430
Load_Register_Immediate header	0x1100_1009	SC	1	0431
3DSTATE_MONOFILTER_SIZE		SC	2	**043C**
3DSTATE_CHROMA_KEY		SC	16	**043E**
NOOP		SC	2	**044E**
NOOP		DM	1	0450
3DSTATE_SAMPLER_PALETTE_LOAD0		DM	257	0451
NOOP		DM	1	0552
3DSTATE_SAMPLER_PALETTE_LOAD1		DM	257	0553
NOOP		DM	12	0654
NOOP		SOL	1	0660
Load_Register_Immediate header	0x1100_1027	SOL	1	0661
SO_NUM_PRIMS_WRITTEN0	0x5200	SOL	4	0662
SO_NUM_PRIMS_WRITTEN1	0x5208	SOL	4	0666
SO_NUM_PRIMS_WRITTEN2	0x5210	SOL	4	066A
SO_NUM_PRIMS_WRITTEN3	0x5218	SOL	4	066E
SO_PRIM_STORAGE_NEEDED0	0x5240	SOL	4	0672
SO_PRIM_STORAGE_NEEDED1	0x5248	SOL	4	0676
SO_PRIM_STORAGE_NEEDED2	0x5250	SOL	4	067A
SO_PRIM_STORAGE_NEEDED3	0x5258	SOL	4	067E
SO_WRITE_OFFSET0	0x5280	SOL	2	0682

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
SO_WRITE_OFFSET1	0x5284	SOL	2	0684
SO_WRITE_OFFSET2	0x5288	SOL	2	0686
SO_WRITE_OFFSET3	0x528C	SOL	2	0688
3DSTATE_SO_BUFFER		SOL	32	068A
NOOP		SOL	3	06AA
3DSTATE_SO_DECL_LIST		SOL	259	06AD
3DSTATE_INDEX_BUFFER		VF	5	07B0
3DSTATE_VERTEX_BUFFERS		VF	133	07B5
3DSTATE_VERTEX_ELEMENTS		VF	69	083A
3DSTATE_VF_STATISTICS		VF	1	087F
3DSTATE_VF		VF	2	0880
3DSTATE_SGVS		VF	2	0882
3DSTATE_VF_INSTANCING		VF	69	0884
3DSTATE_VF_TOPOLOGY		VF	2	08C9
			0	08CB
Load_Register_Immediate header	0x1100_1095	VF	1	08CB
INSTANCE CNT	08300 - 08384h	VF	68	08CC
INSTANCE INDX	08400 - 08484h	VF	68	0910
COMMITTED VERTEX NUMBER	08390h	VF	2	0954
COMMITTED INSTANCE ID	08394h	VF	2	0956
COMMITTED PRIMITIVE ID	08398h	VF	2	0958
STATUS	0839Ch	VF	2	095A
COMMON VERTEX	083A0h	VF	2	095C
				095E
VF_GUARDBAND	083A4h	VF	2	0960
3DSTATE_VF_COMPONENT_PACKING		VF	5	0962
				0967
NOOP		VF	9	0967
Load_Register_Immediate header	0x1100_108D	VFE	1	0970
TDL0 DATA		VFE	142	0971
NOOP		VFE	1	**09FF**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0A00**
TDL1 DATA		VFE	142	**0A01**
NOOP		VFE	1	**0A8F**

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
Load_Register_Immediate header	0x1100_108D	VFE	1	**0A90**
TDL2 DATA		VFE	142	**0A91**
NOOP		VFE	1	**0B1F**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0B20**
TDL3 DATA		VFE	142	**0B21**
NOOP		VFE	1	**0BAF**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0BB0**
TDL4 DATA		VFE	142	**0BB1**
NOOP		VFE	1	**0C3F**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0C40**
TDL5 DATA		VFE	142	**0C41**
NOOP		VFE	1	**0CCF**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0CD0**
TDL6 DATA		VFE	142	**0CD1**
NOOP		VFE	1	**0D5F**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0D60**
TDL7 DATA		VFE	142	**0D61**
NOOP		VFE	1	**0DEF**
Load_Register_Immediate header	0x1100_108D	VFE	1	**0DF0**
TDL8 DATA		VFE	142	**0DF1**
NOOP		VFE	1	**0E7F**
Load_Register_Immediate header	0x1100_1085	VFE	1	**0E80**
GW0 DATA		VFE	134	**0E81**
NOOP		VFE	9	**0F07**
Load_Register_Immediate header	0x1100_1085	VFE	1	**0F10**
GW1 DATA		VFE	134	**0F11**
NOOP		VFE	9	**0F97**
Load_Register_Immediate header	0x1100_1085	VFE	1	**0FA0**
GW2 DATA		VFE	134	**0FA1**
NOOP		VFE	9	**1027**
Load_Register_Immediate header	0x1100_1085	VFE	1	**1030**
GW3 DATA		VFE	134	**1031**
NOOP		VFE	9	**10B7**
Load_Register_Immediate header	0x1100_1085	VFE	1	**10C0**

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
GW4 DATA		VFE	134	**10C1**
NOOP		VFE	9	**1147**
Load_Register_Immediate header	0x1100_1085	VFE	1	**1150**
GW5 DATA		VFE	134	**1151**
NOOP		VFE	9	**11D7**
Load_Register_Immediate header	0x1100_1085	VFE	1	**11E0**
GW6 DATA		VFE	134	**11E1**
NOOP		VFE	9	**1267**
Load_Register_Immediate header	0x1100_1085	VFE	1	**1270**
GW7 DATA		VFE	134	**1271**
NOOP		VFE	9	**12F7**
Load_Register_Immediate header	0x1100_1085	VFE	1	**1300**
GW8 DATA		VFE	134	**1301**
NOOP		VFE	9	**1387**
Load_Register_Immediate header	0x1100_1047	VFE	1	**1390**
GWC DATA		VFE	72	**1391**
NOOP		VFE	71	**13D9**
Load_Register_Immediate header	0x1100_103D	VFE	1	1420
TSG0 DATA		VFE	62	1421
NOOP		VFE	1	145F
Load_Register_Immediate header	0x1100_103D	VFE	1	1460
TSG1 DATA		VFE	62	1461
NOOP		VFE	1	149F
Load_Register_Immediate header	0x1100_103D	VFE	1	14A0
TSG2 DATA		VFE	62	14A1
NOOP		VFE	1	14DF
Load_Register_Immediate header	0x1100_103D	VFE	1	14E0
TSG3 DATA		VFE	62	14E1
NOOP		VFE	1	151F
Load_Register_Immediate header	0x1100_103D	VFE	1	1520
TSG4 DATA		VFE	62	1521
NOOP		VFE	1	155F
Load_Register_Immediate header	0x1100_103D	VFE	1	1560
TSG5 DATA		VFE	62	1561

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
NOOP		VFE	1	159F
Load_Register_Immediate header	0x1100_103D	VFE	1	15A0
TSG6 DATA		VFE	62	15A1
NOOP		VFE	1	15DF
Load_Register_Immediate header	0x1100_103D	VFE	1	15E0
TSG7 DATA		VFE	62	15E1
NOOP		VFE	1	161F
Load_Register_Immediate header	0x1100_103D	VFE	1	1620
TSG8 DATA		VFE	62	1621
NOOP		VFE	1	165F
Load_Register_Immediate header	0x1100_1001/0x1100_1003/0x1100_1009	VFE	1	1660
VFE Data		VFE	2/4/10	1661
NOOP		VFE	8/6/0	1663/1665
Load_Register_Immediate header	0x1100_1001	VFE	1	166B
GP_THREAD_TIME		VFE	2	166C
NOOP		VFE	2	166E
MEDIA_VFE_STATE		VFE	9	1670
MEDIA_POOL_STATE		VFE	6	1679
MEDIA_CURBE_LOAD		VFE	4	167F
MEDIA_INTERFACE_DESCRIPTOR_LOAD		VFE	4	1683
NOOP		VFE	9	1687
NOOP		RS	9	1690
3DSTATE_GATHER_POOL_ALLOC		RS	4	1699
3DSTATE_GATHER_CONSTANT_VS		RS	131	169D
NOOP		RS	13	1720
3DSTATE_GATHER_CONSTANT_GS		RS	131	172D
NOOP		RS	13	17B0
3DSTATE_GATHER_CONSTANT_HS		RS	131	17BD
NOOP		RS	13	1840
3DSTATE_GATHER_CONSTANT_DS		RS	131	184D
NOOP		RS	13	18D0
3DSTATE_GATHER_CONSTANT_PS		RS	131	18DD

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
NOOP		RS	2	1960
3DSTATE_BINDING_TABLE_POOL_ALLOC		RS	4	1962
3DSTATE_BINDING_TABLE_EDIT_VS		RS	258	1966
NOOP		RS	6	1A68
3DSTATE_BINDING_TABLE_EDIT_GS		RS	258	1A6E
NOOP		RS	6	1B70
3DSTATE_BINDING_TABLE_EDIT_HS		RS	258	1B76
NOOP		RS	6	1C78
3DSTATE_BINDING_TABLE_EDIT_DS		RS	258	1C7E
NOOP		RS	6	1D80
3DSTATE_BINDING_TABLE_EDIT_PS		RS	258	1D86
NOOP		RS	4	1E88
MI_BATCH_BUFFER_END/NOOP ***		RS	1	1E8C
NOOP		RS	5	1E8D
3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC		RS	4	1E92
3DSTATE_DX9_CONSTANTF_VS(Global)		RS	1026	1E96
NOOP		RS	6	2298
3DSTATE_DX9_CONSTANTI_VS(Global)		RS	130	229E
NOOP		RS	6	2320
3DSTATE_DX9_CONSTANTB_VS(Global)		RS	18	2326
NOOP		RS	6	2338
3DSTATE_DX9_CONSTANTF_VS(local)		RS	1026	233E
NOOP		RS	6	2740
3DSTATE_DX9_CONSTANTI_VS(local)		RS	130	2746
NOOP		RS	6	27C8
3DSTATE_DX9_CONSTANTB_VS(local)		RS	18	27CE
NOOP		RS	3	27E0
3DSTATE_DX9_LOCAL_VALID_VS		RS	11	27E3
3DSTATE_DX9_CONSTANTF_PS(Global)		RS	1026	27EE
NOOP		RS	6	2BF0
3DSTATE_DX9_CONSTANTI_PS(Global)		RS	130	2BF6
NOOP		RS	6	2C78
3DSTATE_DX9_CONSTANTB_PS(Global)		RS	18	2C7E
NOOP		RS	6	2C90

Description	MMIO Offset/Command	Unit	# of DW	Address Offset (DWord)
3DSTATE_DX9_CONSTANTF_PS(local)		RS	1026	2C96
NOOP		RS	6	3098
3DSTATE_DX9_CONSTANTI_PS(local)		RS	130	309E
NOOP		RS	6	3120
3DSTATE_DX9_CONSTANTB_PS(local)		RS	18	3126
3DSTATE_DX9_LOCAL_VALID_PS		RS	11	3138
3DSTATE_GENERATE_ACTIVE_VS		RS	2	3143
3DSTATE_GENERATE_ACTIVE_PS		RS	2	3145
MI_BATCH_BUFFER_END		RS	1	3147
NOOP		RS	8	3148
URB_ATOMIC_STORAGE		GAFS	8192	3150
				5150
			DW	20816
			Kbytes	81.3125

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 via the MI_FLUSH command immediately before switching to a different pipeline via use of the PIPELINE_SELECT command. Refer to Fixed and Shared Function IDs 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_SAMPLER_PALETTE_LOAD
- 3DSTATE_CHROMA_KEY

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

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

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.

Resource Streamer

This section contains status registers and controls for the resource streamer.

Register
RS_PREEMPT_STATUS_UDW
RS_PREEMPT_STATUS - Resource Streamer Preemption Status
MI_RS_CONTEXT
MI_RS_CONTROL
MI_RS_STORE_DATA_IMM

Resource Streamer Sync Commands

Below is a table of commands that cause the resource streamer to stop and wait until the render command streamer restarts the resource streamer. If a command does not end the current batch buffer or disable the resource streamer, then the command streamer will restart the resource streamer before the next command that is used by the resource streamer.

Resource Streamer Sync Commands: Commands that RS Stops
MI_WAIT_FOR_EVENT
MI_RS_CONTROL
MI_BATCH_BUFFER_END
MI_SEMAPHORE_WAIT
MI_SET_CONTEXT
MI_RS_CONTEXT
MI_BATCH_BUFFER_START
MI_CONDITIONAL_BATCH_BUFFER_END

Introduction

The resource streamer is added to offload work from the driver without compromising on GPU optimizations. In order to reduce latency associated with these offloaded operation, H/W adds a Resource Streamer. The Resource Streamer is almost S/W invisible; S/W sees a single command stream, but it may be best for the S/W to be aware that the RS is present, as certain operations might be emphasized. The resource streamer will run ahead of the 3D Command Streamer and process only the certain commands. The Cmd steamer processes these same commands for purposes of buffer full synchronization and buffer consumption.

Glossary

No special terms identified at this time.

Common Abbreviations

Abbr.	Description/Definition
CS	**Command Streamer.** Block in charge of streaming commands. The Resource Streamer (RS) is primarily an accelerator for the CS.
FF	**Fixed Function.** Any fixed function hardward.
RS	**The Resource Streamer.** Responsible for reducing command latencies for certain command operations.
URB	**Unified Return Buffer.** The mechanism for returning information from a command.

Theory of Operation

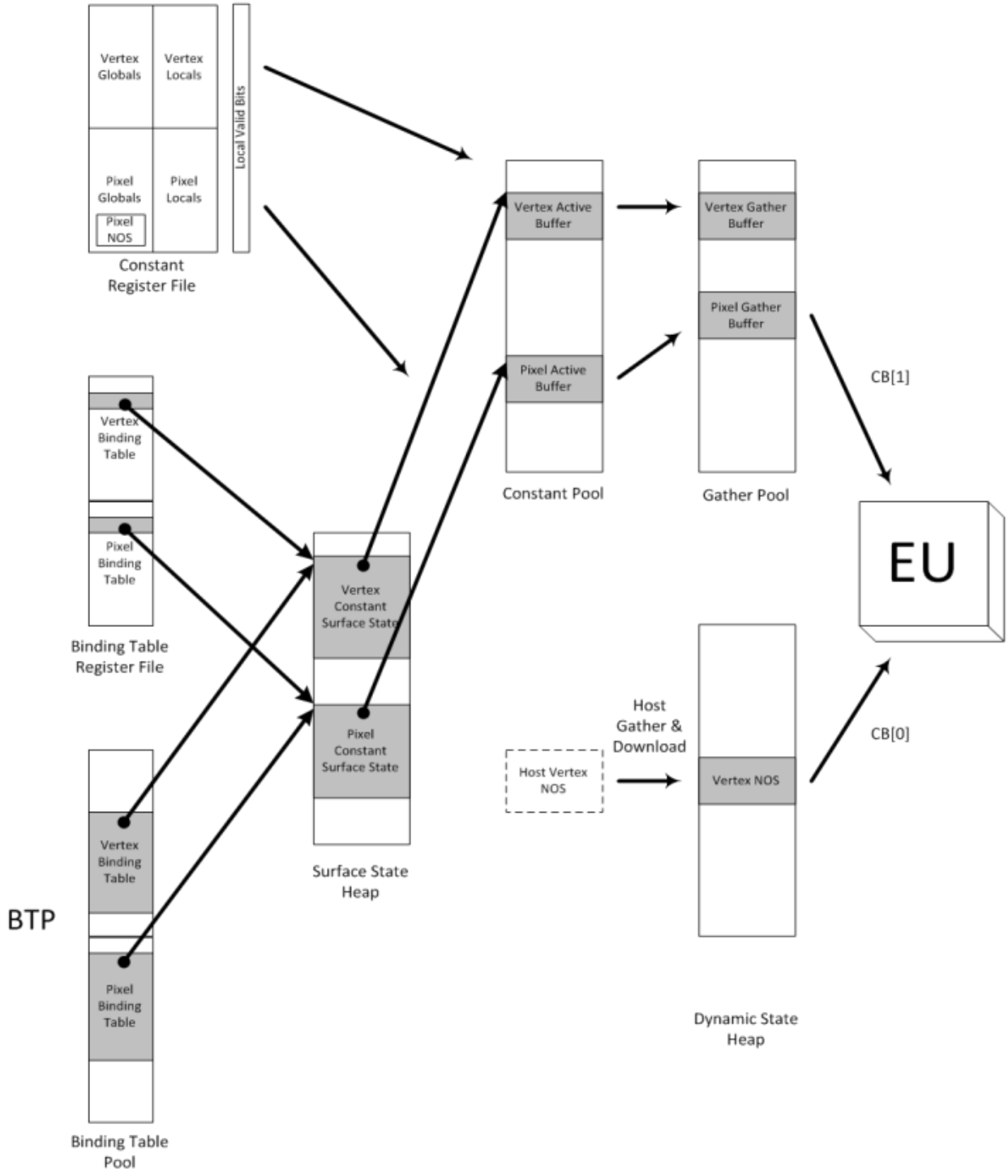
This section briefly describes the operation of the Resource Streamer. Specifically, it calls out reset state, initialization requirements, and major operational tasks of the RS.

Resource Streamer Functions

The Resource Streamer (RS) examines the commands in the ring buffer in an attempt to pre-process certain long latency items for the remainder of the graphics processing. The RS is used for the following operations:

- Batch Processing – The resource streamer reads ahead of command streamer activity to unwind batch buffers.
- Context Save – When the Command Streamer signals that context must be saved, the RS makes certain all previous cycles are completed, saves all context, and signals completion to the command streamer.
- Gather Push Constants – The RS detects GATHER commands (**3DSTATE_GATHER_POOL_ALLOC, 3DSTATE_GATHER_CONSTANT_***), and prefetchs contents needed for further command processing. The RS gets the base address of the contents by detecting the **3DSTATE_GATHER_POOL_ALLOC** command, and uses other **3DSTATE_GATHER_CONSTANT_*** commands to generate reads for data, and writes out data to memory.
- Constant Buffer Generation – Similar to other constant processes, the RS intercepts the commands for constants to update state and data.
- HW Binding Table Generation/Flush – The RS detects operations in the command stream to update binding table state and memory with bind table contents.

Resource Streamer Activity Diagram



Detailed Resource Streamer Operations

Introduction

This chapter describes the operation of the Resource Streamer in deeper detail. Most of the operations of the Resource Streamer are processed from ring buffer shown in the Ring Buffer Organization Figure in Resource Streamer Operation Descriptions. The RS examines the command stream from the ring buffer to pre-process information required by the 3D Command Streamer (CS). For a large number of the commands, the RS takes no action.

Resource Streamer Operation Descriptions

Batch Processing

When an MI_BATCH_BUFFER_START command is parsed by the render command streamer and the resource streamer enable bit is set, the command stream flags that the resource streamer is enabled. Once it parses a non-sync command then it sends the current address for where the RS must start parsing the batch buffer. The Resource Streamer parses commands until it parses a sync command, which causes the resource streamer to send a message to the command streamer that it has stopped; RS then goes idle. The command streamer stalls at a sync command until the resource streamer specifies it has stopped, and restarts the resource streamer at the next non-sync command. Below is a link to the topic with sync commands.

Commands Actions in the RS

Context Save

When the CS indicates that there is a context to be saved or restored, the RS saves its context. The CS provides an address for the RS image and issues a “batch buffer start” (see section *Batch Processing*). The RS consumes this image like any other batch buffer, and stops when it reaches the MI_BATCH_BUFFER_END command.

The context image for the Resource streamer consists of the following components:

1. HW_BINDING_TABLE_IMAGE
2. GATHER_IMAGE
3. CONSTANT_IMAGE
4. MI_BATCH_BUFFER_END

These are discussed in the following subsections.

HW Binding Table Image

While it is not always necessary to save binding table information, “split points” context switches must be saved, so the binding table contents are always saved. These consist of:

- Binding Table Generate Enable
- Binding Table Pool Base Address
- Binding Table Pool Size
- Binding Table Contents

HW Binding Table Image

Description	Dwords Required for Storage
3DSTATE_BINDING_TABLE_POOL_ALLOC	3
3DSTATE_BINDING_TABLE_EDIT_VS	194
3DSTATE_BINDING_TABLE_EDIT_GS	194
3DSTATE_BINDING_TABLE_EDIT_HS	194
3DSTATE_BINDING_TABLE_EDIT_DS	194
3DSTATE_BINDING_TABLE_EDIT_PS	194
3DSTATE_BINDING_TABLE_EDIT_VS	194

Gather Push Constants Image

Since the resource streamer does not support mid-triangle preemption, the resource steamer will have finished producing all the gather buffers by the end of the batch buffer and the cmd streamer would have consumed all the gather buffers. The following things need to be saved.

- Gather pool enable
- Gather pool base address
- Gather pool size

Therefore a 3DSTATE_GATHER_POOL_ALLOC command needs to be saved.

Gather Push Constants Image

Description	Dwords Required for Storage
3DSTATE_GATHER_POOL_ALLOC	4

Push Constant Image

We assume that the end of the batch buffer can come between any set of cmds. Therefore the following things will be saved:

- Dx9 Constant enable
- Dx9 Constant pool base address
- Dx9 Constant pool size
- Dx9 local registers (F,I,B)
- Dx9 Local Valid
- Dx9 global registers (F,I,B)

Therefore a 3DSTATE_CONSTANT_BUFFER_POOL_ALLOC command will saved. In addition, since the F register is 256 entries and only a maximum of 63 entries can be contained in a single 3DSTATE_DX9CONSTANTF_* command, 5 CONSTANTF cmds will be saved for global and 5 for local registers register per FF (VS,PS). There will be 1 3DSTATE_CONSTANTI_* will be save for global and 1 for local register per FF. There will be 1 3DSTATE_CONSTANTB_* will be save for global and 1 for local register per FF.

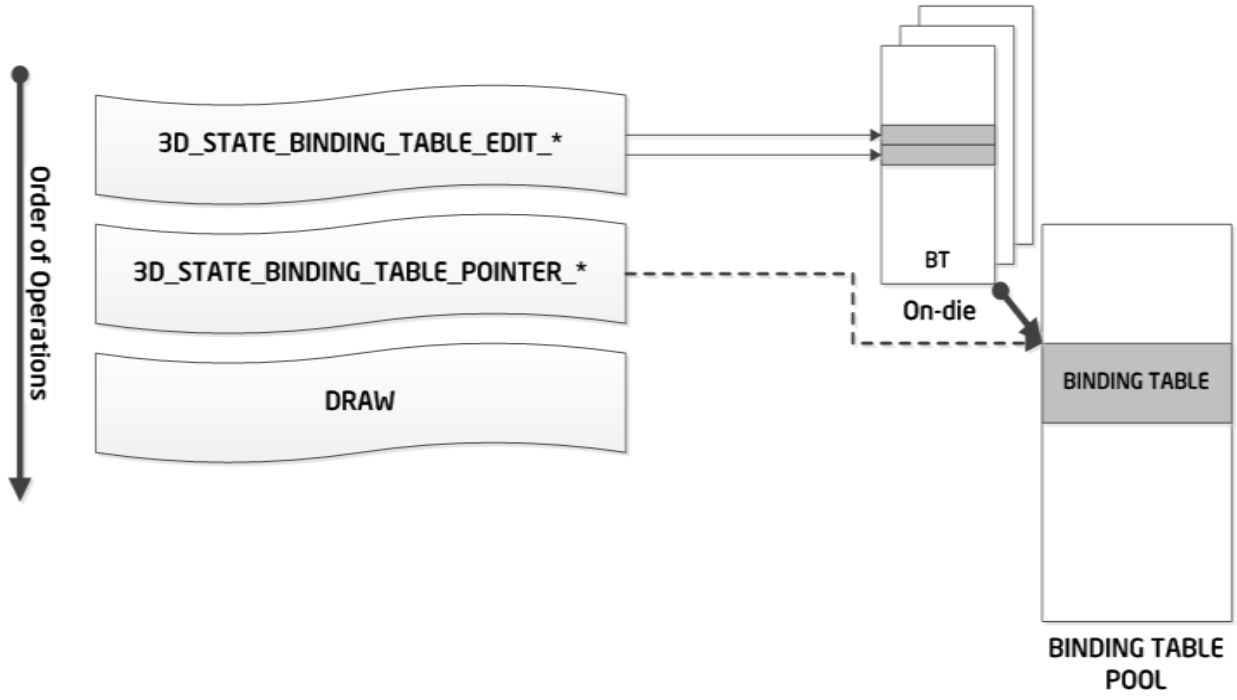
Gather Push Constants Image

Description	Dwords Required for Storage
3DSTATE_CONSTANT_BUFFER_POOL_ALLOC	4
3DSTATE_CONSTANTF_VS	1026
3DSTATE_CONSTANTI_VS	130
3DSTATE_CONSTANTB_VS	18
3DSTATE_CONSTANTF_VS	1026
3DSTATE_CONSTANTI_VS	130
3DSTATE_CONSTANTB_VS	18
3DSTATE_LOCAL_VALID_VS	10
3DSTATE_CONSTANTF_PS	1026
3DSTATE_CONSTANTI_PS	130
3DSTATE_CONSTANTB_PS	18
3DSTATE_CONSTANTF_PS	1026
3DSTATE_CONSTANTI_PS	130
3DSTATE_CONSTANTB_PS	18
3DSTATE_LOCAL_VALID_PS	10

HW Binding Table Generation

The RS generates binding tables in hardware to offload this from the driver. There is an on-die set of binding tables for each fixed-function unit (VS, GS, HS, DS, and PS). There is a set of commands generated by the driver to update each of these tables (3D_STATE_BINDING_TABLE_POINTER_*). When the RS encounters any of these commands, it writes the corresponding binding table out to the binding table pool. When the CS encounters these commands, it sends the binding table points down as pipelined state.

HW Binding Table Generation



The following table describes the different types of usages with binding table generation.

RS Active *	BT Pool Enabled	Mode
0	0	SW Generate BT in Surface State Heap
1	0	Illegal **
1	1	HW Generate BT

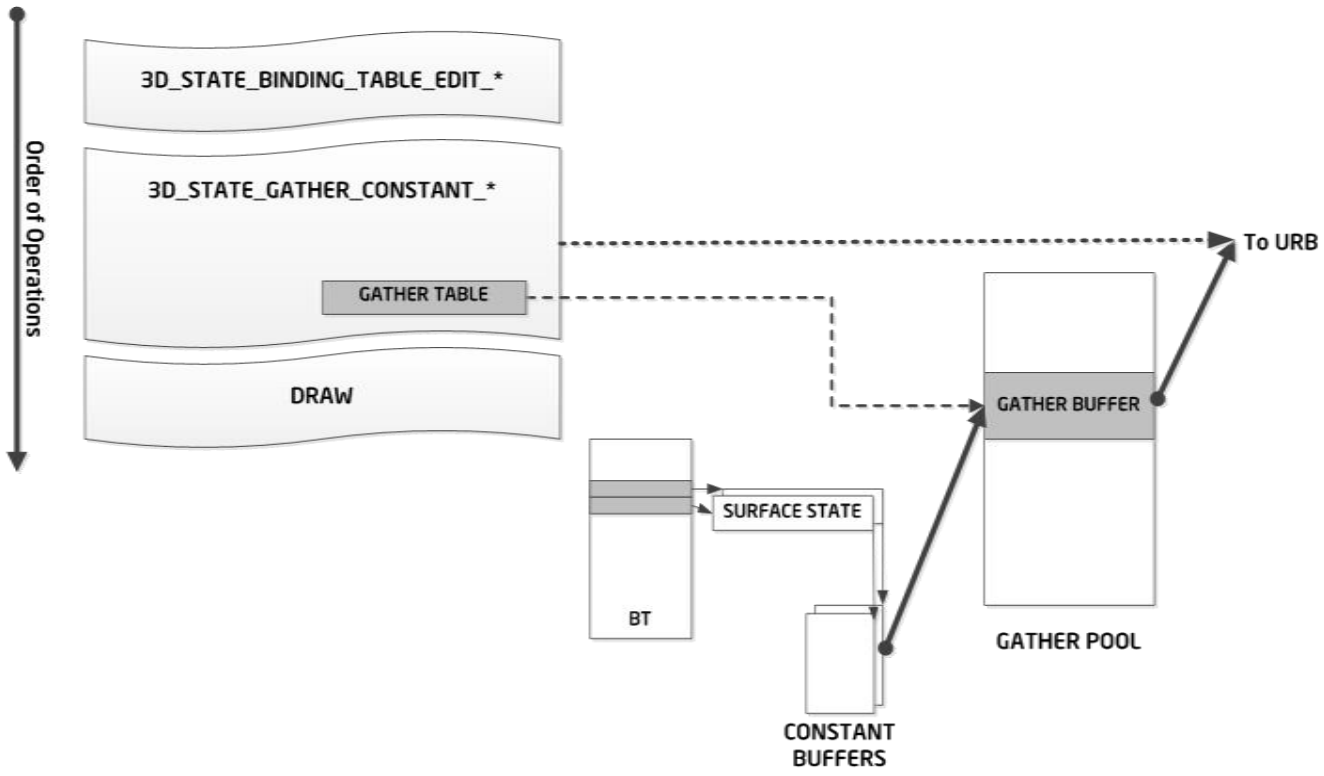
Gather Push Constants

Applications can provide up to 16 constant buffers. The compiler does some optimizations of constant usage and determines which constants should be packed in which constant register for optimal shader performance. While this gathering and packing of constant elements into push constants optimizes the shaders, it causes the driver additional work at draw call time, since the driver must gather and pack the constants at draw time.

The RS offloads the gathering process for the driver by interpreting the 3D_STATE_GATHER_CONSTANT_* for each of the fixed functions (VS, GS, DS, HS, PS). The compiler generates at gather table which instructs which elements of the buffers should be packed into the gather buffer. The gather table indexes the binding table to get a surface state which in turn points to the constant buffer. Once the gather buffer has been filled, the CS will execute the 3D_STATE_GATHER_CONSTANT_* to load the push constant into the URB.

Note: The gather push constants can ONLY BE USED if the HW generated binding tables are also used.

Gather Push Constants Generation

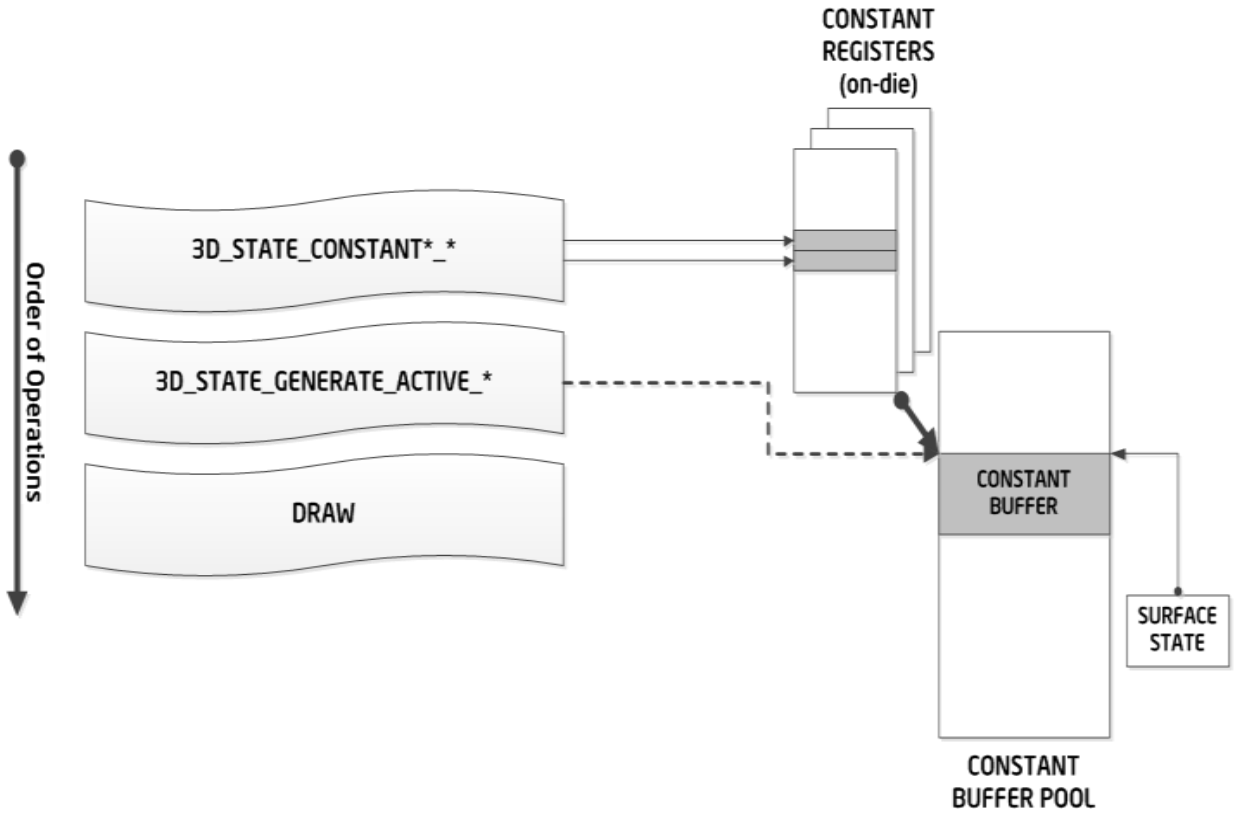


Constant Buffer Generation (not DX9)

The constant model used is a set of registers that the application can incrementally update. The hardware requires a constant buffer which lives until the last shader using that buffer retires. To offload the driver the 3D_STATE_CONSTANT*_* commands are used. The constant registers can be either floating, integer, or Boolean (signified by the commands CONSTANTF, CONSTANTI, or CONSTANTB, respectively). The option determines the fixed function for the constants (VS, GS, DS, HS, or PS).

When all edits to the constant registers have been completed, the 3D_STATE_GENERATE_ACTIVE_* command is used to write out a constant buffer to the Constant Buffer Pool. These buffers are fixed at 8Kbytes. Software is required to provide a surface state object that points to the constant buffer created.

Constant Buffer Generation



Commands Actions in the RS

The tables below show all 3D commands processed by the RS. In the following tables, "STOP" indicates that the RS waits for all engines to complete operations AND invalidates all command data currently in the command FIFO. "BLOCK" indicates that the RS waits for all engines to complete operation, stops further command parsing, but retains data in the command FIFO.

MI Commands Processing in the RS

Opcode (28:23)	Command	RS Handing (No Perf)	RS Handling (Perf)	Notes
03h	MI_WAIT_FOR_EVENT	STOP	BLOCK	
05h	MI_ARB_CHECK	STOP	STOP	
06h	MI_RS_CONTROL	STOP	STOP	
0Ah	MI_BATCH_BUFFER_END	STOP	STOP	
16h	MI_SEMAPHORE_MBOX	STOP	BLOCK	
18h	MI_SET_CONTEXT	STOP	STOP	
1Ah	MI_RS_CONTEXT	STOP	STOP	
31h	MI_BATCH_BUFFER_START	STOP	STOP	
36h	MI_CONDITIONAL_BATCH_BUFFER_END	STOP	STOP	

Other Commands Processed in the RS

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
0h	1h	01h	STATE_BASE_ADDRESS	RS LATCH	RS LATCH	RSunit updates the state base address if parsed
1h	1h	04h	PIPELINE_SELECT	STOP	STOP	Stop only if 3D is not selected
3h	0h	03h	Reserved			
3h	0h	04h	3DSTATE_CLEAR_PARAMS			Refer to 3D Pipeline
3h	0h	05h	3DSTATE_DEPTH_BUFFER			Refer to 3D Pipeline
3h	0h	06h	Reserved			
3h	0h	06h	3DSTATE_STENCIL_BUFFER			Refer to 3D Pipeline
3h	0h	07h	Reserved			
3h	0h	07h	3DSTATE_HIER_DEPTH_BUFFER			Refer to 3D Pipeline
3h	0h	08h	3DSTATE_VERTEX_BUFFERS			Refer to Vertex Fetch
3h	0h	09h	3DSTATE_VERTEX_ELEMENTS			Refer to Vertex Fetch
3h	0h	0Ah	3DSTATE_INDEX_BUFFER			Refer to Vertex Fetch
3h	0h	0Bh	3DSTATE_VF_STATISTICS			Refer to Vertex Fetch
3h	0h	0Ch	Reserved			
3h	0h	0Dh	3DSTATE_VIEWPORT_STATE_POINTERS			Refer to 3D Pipeline
3h	0h	10h	3DSTATE_VS			Refer to Vertex Shader
3h	0h	11h	3DSTATE_GS			Refer to Geometry Shader

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
3h	0h	12h	3DSTATE_CLIP			Refer to Clipper
3h	0h	13h	3DSTATE_SF			Refer to Strips and Fans
3h	0h	14h	3DSTATE_WM			Refer to Windower
3h	0h	15h	3DSTATE_CONSTANT_VS			Refer to Vertex Shader
3h	0h	16h	3DSTATE_CONSTANT_GS			Refer to Geometry Shader
3h	0h	17h	3DSTATE_CONSTANT_PS			Refer to Windower
3h	0h	18h	3DSTATE_SAMPLE_MASK			Refer to Windower
3h	0h	19h	3DSTATE_CONSTANT_HS			Refer to Hull Shader
3h	0h	1Ah	3DSTATE_CONSTANT_DS			Refer to Domain Shader
3h	0h	1Bh	3DSTATE_HS			Refer to Hull Shader
3h	0h	1Ch	3DSTATE_TE			Refer to Tesselator
3h	0h	1Dh	3DSTATE_DS			Refer to Domain Shader
3h	0h	1Eh	3DSTATE_STREAMOUT			Refer to HW Streamout
3h	0h	1Fh	3DSTATE_SBE			Refer to Setup
3h	0h	20h	3DSTATE_PS			Refer to Pixel Shader
3h	0h	21h	Reserved			
3h	0h	22h	3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP			Refer to Strips & Fans

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
3h	0h	23h	3DSTATE_VIEWPORT_STATE_POINTERS_CC			Refer to Windower
3h	0h	24h	3DSTATE_BLEND_STATE_POINTERS			Refer to Pixel Shader
3h	0h	25h	3DSTATE_DEPTH_STENCIL_STATE_POINTERS			Refer to Pixel Shader
3h	0h	26h	3DSTATE_BINDING_TABLE_POINTERS_VS	Generate BT if HW BT enabled	Generate BT if HW BT enabled	
3h	0h	27h	3DSTATE_BINDING_TABLE_POINTERS_HS	Generate BT if HW BT enabled	Generate BT if HW BT enabled	
3h	0h	28h	3DSTATE_BINDING_TABLE_POINTERS_DS	Generate BT if HW BT enabled	Generate BT if HW BT enabled	
3h	0h	29h	3DSTATE_BINDING_TABLE_POINTERS_GS	Generate BT if HW BT enabled	Generate BT if HW BT enabled	
3h	0h	2Ah	3DSTATE_BINDING_TABLE_POINTERS_PS	Generate BT if HW BT enabled	Generate BT if HW BT enabled	
3h	0h	2Fh	Reserved			
3h	0h	30h	3DSTATE_URB_VS/td>	Execute	Execute	
3h	0h	31h	3DSTATE_URB_HS	Execute	Execute	
3h	0h	32h	3DSTATE_URB_DS	Execute	Execute	
3h	0h	33h	3DSTATE_URB_GS	Execute	Execute	
3h	0h	34h	3DSTATE_GATHER_VS	Execute	Execute	
3h	0h	35h	3DSTATE_GATHER_GS	Execute	Execute	
3h	0h	36h	3DSTATE_GATHER_HS	Execute	Execute	
3h	0h	37h	3DSTATE_GATHER_DS	Execute	Execute	
3h	0h	38h	3DSTATE_GATHER_PS	Execute	Execute	
3h	0h	39h	3DSTATE_CONSTANTF_VS	Execute	Execute	
3h	0h	3Ah	3DSTATE_CONSTANTF_PS	Execute	Execute	

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
3h	0h	3Bh	3DSTATE_CONSTANTI_VS	Execute	Execute	
3h	0h	3Ch	3DSTATE_CONSTANTI_PS	Execute	Execute	
3h	0h	3Dh	3DSTATE_CONSTANTB_VS	Execute	Execute	
3h	0h	3Eh	3DSTATE_CONSTANTB_PS	Execute	Execute	
3h	0h	3Fh	3DSTATE_LOCAL_VALID_VS	Execute	Execute	
3h	0h	40h	3DSTATE_LOCAL_VALID_PS	Execute	Execute	
3h	0h	41h	3DSTATE_GENERATE_ACTIVE_VS	Execute	Execute	
3h	0h	42h	3DSTATE_GENERATE_ACTIVE_PS	Execute	Execute	
3h	0h	43h	3DSTATE_BINDING_TABLE_EDIT_VS			Refer to Vertex Shader
3h	0h	44h	3DSTATE_BINDING_TABLE_EDIT_GS			Refer to Vertex Shader
3h	0h	45h	3DSTATE_BINDING_TABLE_EDIT_HS			Refer to Vertex Shader
3h	0h	46h	3DSTATE_BINDING_TABLE_EDIT_DS			Refer to Vertex Shader
3h	0h	47h	3DSTATE_BINDING_TABLE_EDIT_PS			Refer to Vertex Shader
3h	0h	48h	3DSTATE_VF_HASHING			
3h	0h	49h	3DSTATE_VF_INSTANCING			
3h	0h	4Ah	3DSTATE_VF_SGVS			
3h	0h	4Bh	3DSTATE_VF_TOPOLOGY			
3h	0h	4Ch	3DSTATE_WM_CHROMA_KEY			
3h	0h	4Dh	3DSTATE_PS_BLEND			
3h	0h	4Eh	3DSTATE_WM_DEPTH_STENCIL			
3h	0h	4Fh	3DSTATE_PS_EXTRA			
3h	0h	50h	3DSTATE_RASTER			
3h	0h	51h	3DSTATE_SBE_SWIZ			
3h	0h	52h	3DSTATE_WM_HZ_OP			
3h	0h	53h	3DSTATE_INT (internally generated state)			
3h	0h	54h	3DSTATE_RS_CONSTANT_POINTER			

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
3h	0h	55h	3DSTATE_VF_COMPONENT_PACKING			
3h	0h	57h	3DSTATE_DX9_CONSTANTF_HS			
3h	0h	58h	3DSTATE_DX9_CONSTANTI_HS			
3h	0h	59h	3DSTATE_DX9_CONSTANTB_HS			
3h	0h	5ah	3DSTATE_DX9_LOCAL_VALID_HS			
3h	0h	5bh	3DSTATE_DX9_GENERATE_ACTIVE_HS			
3h	0h	5ch	3DSTATE_DX9_CONSTANTF_DS			
3h	0h	5dh	3DSTATE_DX9_CONSTANTI_DS			
3h	0h	5eh	3DSTATE_DX9_CONSTANTB_DS			
3h	0h	5fh	3DSTATE_DX9_LOCAL_VALID_DS			
3h	0h	60h	3DSTATE_DX9_GENERATE_ACTIVE_DS			
3h	0h	61h	3DSTATE_DX9_CONSTANTF_GS			
3h	0h	62h	3DSTATE_DX9_CONSTANTI_GS			
3h	0h	63h	3DSTATE_DX9_CONSTANTB_GS			
3h	0h	64h	3DSTATE_DX9_LOCAL_VALID_GS			
3h	0h	65h	3DSTATE_DX9_GENERATE_ACTIVE_GS			
3h	0h	67h-FFh	Reserved			
3h	1h	00h	3DSTATE_DRAWING_RECTANGLE			
3h	1h	02h	3DSTATE_SAMPLER_PALETTE_LOAD0			
3h	1h	03h	Reserved			
3h	1h	04h	3DSTATE_CHROMA_KEY			
3h	1h	05h	Reserved			
3h	1h	06h	3DSTATE_POLY_STIPPLE_OFFSET			
3h	1h	07h	3DSTATE_POLY_STIPPLE_PATTERN			
3h	1h	08h	3DSTATE_LINE_STIPPLE			
3h	1h	0Ah	3DSTATE_AA_LINE_PARAMS			
3h	1h	0Bh	3DSTATE_GS_SVB_INDEX			
3h	1h	0Ch	3DSTATE_SAMPLER_PALETTE_LOAD1			
3h	1h	0Dh	3DSTATE_MULTISAMPLE			
3h	1h	0Eh	3DSTATE_STENCIL_BUFFER			
3h	1h	0Fh	3DSTATE_HIER_DEPTH_BUFFER			
3h	1h	10h	3DSTATE_CLEAR_PARAMS			
3h	1h	11h	3DSTATE_MONOFILTER_SIZE			
3h	1h	12h	3DSTATE_PUSH_CONSTANT_ALLOC_VS			

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
3h	1h	13h	3DSTATE_PUSH_CONSTANT_ALLOC_HS			
3h	1h	14h	3DSTATE_PUSH_CONSTANT_ALLOC_DS			
3h	1h	15h	3DSTATE_PUSH_CONSTANT_ALLOC_GS			
3h	1h	16h	3DSTATE_PUSH_CONSTANT_ALLOC_PS			
3h	1h	17h	3DSTATE_SO_DECL_LIST			
3h	1h	18h	3DSTATE_SO_BUFFER			
3h	1h	19h	3DSTATE_BINDING_TABLE_POOL_ALLOC			
3h	1h	1Ah	3DSTATE_GATHER_POOL_ALLOC			
3h	1h	1Bh	3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC			
3h	1h	1Ch	3DSTATE_SAMPLE_PATTERN			
3h	1h	1Dh-FFh	Reserved			
3h	1h	19h	3DSTATE_BINDING_TABLE_POOL_ALLOC	Execute	Execute	
3h	1h	1Ah	3DSTATE_GATHER_POOL_ALLOC	Execute	Execute	
3h	1h	1Bh	3DSTATE_CONSTANT_BUFFER_POOL_ALLOC	Execute	Execute	
3h	1h	1Ch	3DSTATE_SAMPLE_PATTERN			
3h	1h	1Dh-FFh	Reserved			
3h	2h	00h	PIPE_CONTROL			
3h	2h	01h-FFh	Reserved			
3h	3h	00h	3DPRIMITIVE	Sync	Sync	3DPRIMITIVE command is unique in that it tells the engines to send fence cycles, but does not stop RSunit (not a sync point)

Resource Streamer Programming Guidelines

This section describes RS activities and assumptions that are required for programming.

RS Interactions with the 3D Command Streamer

Because the Resource Streamer is processing ahead of the Command Streamer, many of the commands interpreted by the RS are a signal to stop further processing. In these cases, the RS completes pending activity, and waits for an indication from the Command Streamer to start again.

The specific cases that the CS commands the RS to continue are:

- Batch Buffer command parsing
- Context save

RS Interactions with Memory Requests

The RS is responsible for the generation of a number of memory requests. These are:

- Make batch buffer read requests (when address is supplied from the CS).
- Make push constant gather read requests from the state base offset.
- Make push constant gather write of packed data to the gather pool.
- Fetch the gather buffer surface base address.
- Write out the binding table pointer (BTP).
- Save BTP, constant buffer, and gather constant context data to an offset into the context image.
- Write out constant data.

As is the case in all memory accesses, read requests from the RS can be freely reordered, and may be returned in any order by the hardware. The RS consumes the cycles and presents the “software” order transparently.

When accessing the same address, a write operation followed by the read returns the written data. Writes to non-overlapping addresses may be freely reordered as well. Fencing is used to make certain all writes up to the fence have completed.

Fundamental Programming and Operational Assumptions

The following assumptions are made in the RS, and these are useful limitations to the programming:

- The CS can never send a request to a busy RS. The RS will have foreseen the situation, and stopped its operations before the CS action.
- Surface base address is never changed while in a batch buffer.
- Push constant data is 128-bit aligned.
- The GATHER command should have Constant Buffer valid bits set for any indices used in the command.

Non-Operational Activities

There are no specific events or performance counters for the resource streamer (RS).

Hardware Binding Tables

The driver spends a considerable amount of time managing the binding tables. A new command is added, 3DSTATE_BINDING_TABLE_EDIT_*, to offload the binding table generation from the driver. There is an on-die set of binding tables for each FF (VS, GS, HS, DS, PS). The 3DSTATE_BINDING_TABLE_EDIT_* commands are used by the driver to update these tables. The 3DSTATE_BINDING_TABLE_POINTER_* commands are added. When the resource streamer encounters a 3DSTATE_BINDING_TABLE_POINTER_* command, it writes the binding table out to the binding table pool. When the command streamer encounters a 3DSTATE_BINDING_TABLE_POINTER_* command, it sends the binding table pointer down as pipelined state.

Hardware Binding Tables are only supported for 3D workloads. The resource streamer must be enabled only for 3D workloads. The resource streamer must be disabled for Media and GPGPU workloads. A batch buffer containing both 3D and GPGPU workloads must take care of disabling and enabling the Resource Streamer appropriately while changing the PIPELINE_SELECT mode from 3D to GPGPU and vice versa. The resource streamer must be disabled using MI_RS_CONTROL command and Hardware Binding Tables must be disabled by programming 3DSTATE_BINDING_TABLE_POOL_ALLOC with "Binding Table Pool Enable" set to disable (i.e value '0'). The following example shows disabling and enabling of the resource streamer in a batch buffer for 3D and GPGPU workloads:

```
MI_BATCH_BUFFER_START (Resource Streamer Enabled)
PIPELINE_SELECT (3D)
3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Enabled)
3D WORKLOAD MI_RS_CONTROL (Disable Resource Streamer)
3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Disabled)
PIPELINE_SELECT (GPGPU)
GPGPU Workload
PIPELINE_SELECT (3D)
MI_RS_CONTROL (Enable Resource Streamer)
3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Enabled)
3D WORKLOAD
MI_BATCH_BUFFER_END
```

3DSTATE_BINDING_TABLE_POOL_ALLOC

Programming Note
The binding table generator feature has a simple all or nothing model. If HW generated binding tables are enabled, the driver must enable the pool and use 3D_HW_BINDING_TABLE_POINTER_* commands.
When switching between HW and SW binding table generation, SW must issue a state cache invalidate.
A maximum of 16,383 Binding Tables are allowed in any batch buffer.

The variable length commands are 3DSTATE_BINDING_TABLE_EDIT_HS, 3DSTATE_BINDING_TABLE_EDIT_DS, and 3DSTATE_BINDING_TABLE_EDIT_PS.

3DSTATE_BINDING_TABLE_POOL_ALLOC

3DSTATE_BINDING_TABLE_EDIT_VS

3DSTATE_BINDING_TABLE_EDIT_HS

3DSTATE_BINDING_TABLE_EDIT_DS

3DSTATE_BINDING_TABLE_EDIT_GS

3DSTATE_BINDING_TABLE_EDIT_PS

Gather Constants

Gather commands support fetching from 16 different constant buffers or one constant buffer of 8KB size. The compiler does some optimizations of constant usage and determines which elements of which constants should be packed in which push constant register for optimum shader performance. While this gathering and packing of constant elements into push constant registers optimizes the shader, it causes the driver added work at draw call time, because the driver must do the gather and packing at draw time. New commands(3D_STATE_GATHER_CONSTANT_* and 3DSTATE_GATHER_POOL_ALLOC) were added to offload the gather and packing functions from the driver. The base address for the push constant buffer and the enabling of the feature is programmed through the 3DSTATE_GATHER_POOL_ALLOC. There are 5 FF which support push constants (VS, GS, DS, HS, PS) and they all have corresponding gather commands. The compiler generates a gather table that specifies what elements of what buffers are packed into the gather buffer. The gather table indexes the BT to get the surface state which points to the constant buffer. The resource streamer gathers constants by reading the constant buffer, packs the data and then writes the buffer out to a push constant buffer based on the base address and the offset in the 3DSTATE_GATHER_CONSTANT_* command.

3DSTATE_GATHER_POOL_ALLOC

3DSTATE_GATHER_CONSTANT_VS

3DSTATE_GATHER_CONSTANT_HS

3DSTATE_GATHER_CONSTANT_DS

3DSTATE_GATHER_CONSTANT_GS

3DSTATE_GATHER_CONSTANT_PS

Workarounds

Workaround
Commands executed after any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer valid greater than zero.

Dx9 Constant Buffer Generation

The Dx9 constant model is a set of register that the App can incrementally update. The HW requires a constant buffer which lives until the last shader using that buffer retires. To offload the driver the 3DSTATE_DX9_CONSTANT*_* cmds are added. These commands allow the on-die constant register to be maintained. When all the edits to the constant register have been completed, the 3DSTATE_DX9_GENERATE_ACTIVE_* cmd is used to write out a constant buffer to the Dx9 Constant buffer pool. The Dx9 constant buffers are fixed 8KB in size, with a large portion of the second 4KB unused.

Programming Note	
Context:	Dx9 Constant Buffer generation
For buffers, which have no inherent "height," padding requirements are different. A buffer must be padded to the next multiple of 256 array elements, with an additional 16 bytes added beyond that to account for the L1 cache line.	

3DSTATE_RS_CONSTANT_POINTER

Programming Note	
Context:	Dx9 Constant Buffer generation.
• The Dx9 constant buffer feature has a simple all or nothing model. • A maximum of 16,383 Binding Tables are allowed in any batch buffer. • The Dx9 constants can only be enabled if the binding table generator is also enabled.	

3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC

Vertex Shader Constant

This section contains various commands for the vertex shader constant.

3DSTATE_DX9_CONSTANTF_VS

3DSTATE_DX9_CONSTANTI_VS

3DSTATE_DX9_CONSTANTB_VS

3DSTATE_DX9_LOCAL_VALID_VS

3DSTATE_DX9_GENERATE_ACTIVE_VS

Pixel Shader Constant

This section contains various commands for the pixel shader constant.

3DSTATE_DX9_CONSTANTF_PS

3DSTATE_DX9_CONSTANTI_PS

3DSTATE_DX9_CONSTANTB_PS

3DSTATE_DX9_LOCAL_VALID_PS

3DSTATE_DX9_GENERATE_ACTIVE_PS

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.

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

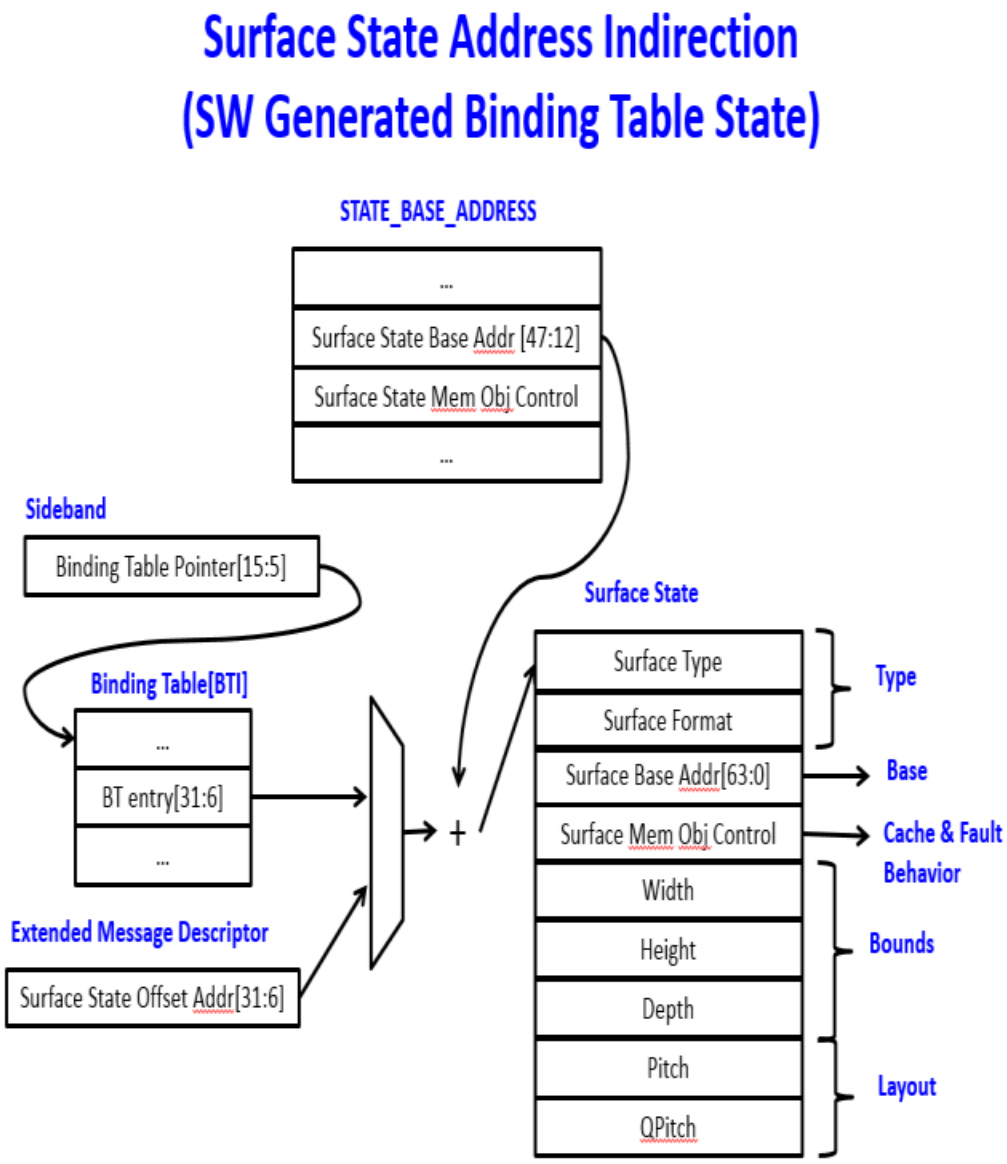
Programming Note	
Context:	Shared Function ID
SFID_DP_DC1 is an extension of SFID_DP_DC0 to allow for more messages types. They act as a single logical entity.	

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.	

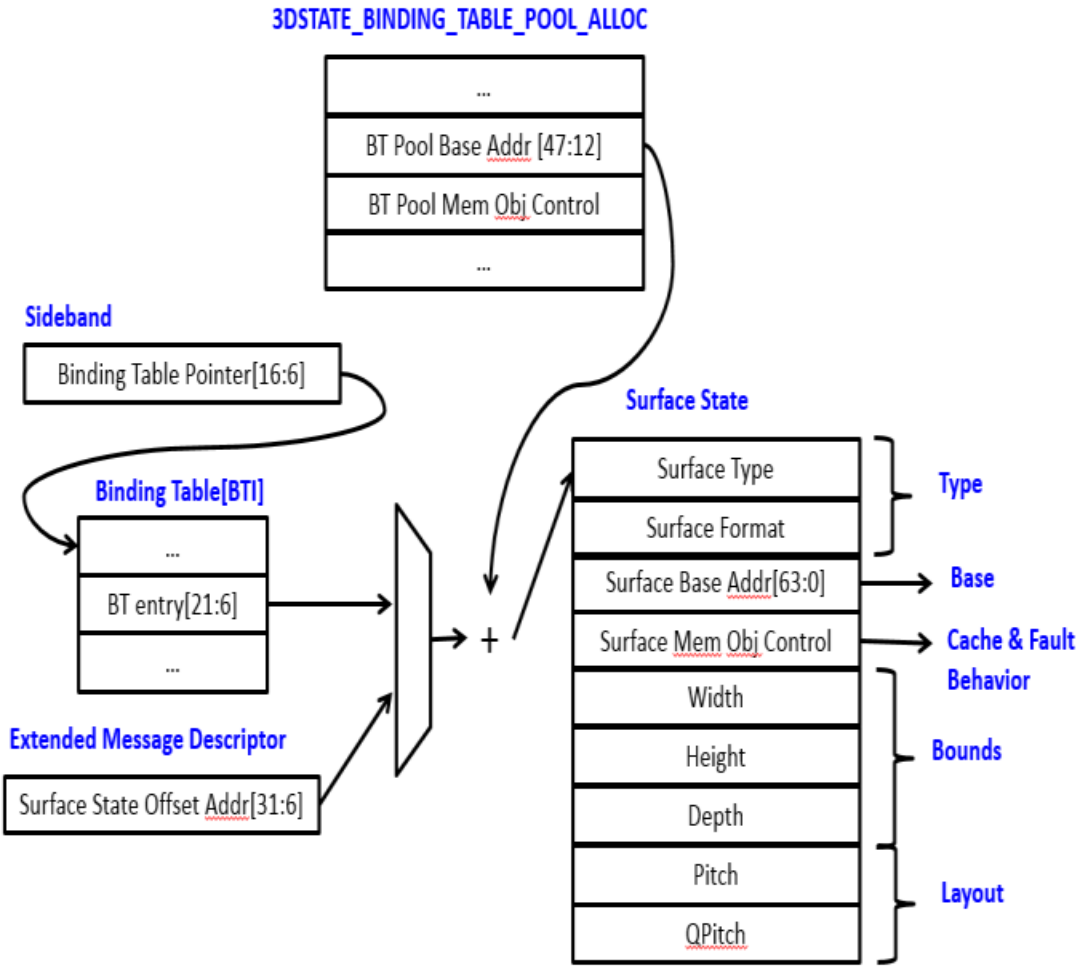
Binding Table

The surface state model is intended to be used for constant buffers, render target surfaces, and media surfaces.

Surface State Model Memory Characteristics



Surface State Address Indirection (HW Generated Binding Table)



Below is a table specifying the different configuratiosn of the Binding Table Pointers and the BINDING_TABLE_STATE.

Binding Table Pool Enable	Binding Table Entry Format	Binding Table Pointer	Base Address
0	32 bit size/[31:6]	15:5	Surface State Base Address
1	16 bit size/[21:6]	16:6	Binding Table Pool Address

Filtered Table (put table title on this line)

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**

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 Gen Core in response to sampling engine messages. The sampling engine uses SAMPLER_STATE to control filtering modes, address control modes, and other features of the sampling engine. A pointer to the sampler state is delivered with each message, and an index selects one of 16 states pointed to by the pointer. Some messages do not require SAMPLER_STATE. In addition, the sampling engine uses SURFACE_STATE to define the attributes of the surface being sampled. This includes the location, size, and format of the surface as well as other attributes.

Although data is commonly used for “texturing” of 3D surfaces, the data can be used for any purpose once returned to the execution core. The 3D Sampler can be used to assist the media sampler in specific operations such as video scaling.

The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the Gen Core to complete the *sample* instruction.

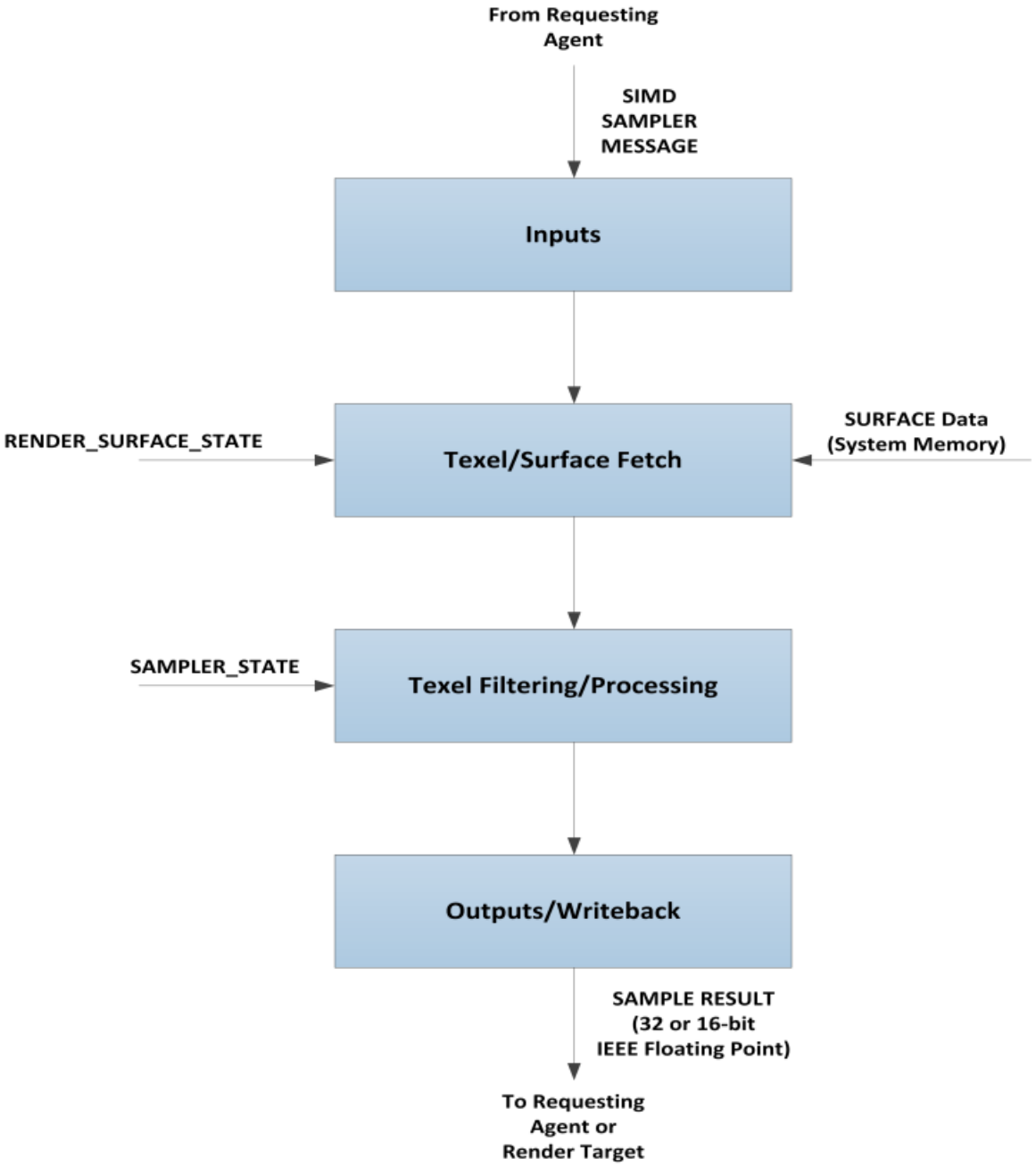
Subfunction	Description
Texture Coordinate Processing	Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include cube map intersection.
Texel Address	The Sampling Engine determines the required set of texel samples (specific texel values from

Subfunction	Description
Generation	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 Graphics Core which receives messages from other agents in the Graphics Core, fetches data from external memory sources typically referred to as "surfaces", performs operations on the data and returns the results in standard formats to the requester (or directly to a Render Target is requested). One of the most common applications of the 3D sampler is to return a filtered/blended pixel from a location in a texture map.



Sampler Inputs Messages

Input requests to the 3D Sampler are in the form of messages (see Messages sub-section for a description of message types and formats). A pixel shader kernal executing on the Graphics Core is an example of an agent which is capable of sending sample messages to the 3D Sampler.

In its most basic form, the sampler receives coordinates to a location within a field of data (often a texture map or depth map) and returns a value which represents the pixel color or depth which may be

filtered/blended as defined by associated surface and sampler state objects. Sampler can also work on un-typed data structures called buffers.

Messages are sent in SIMD (Single Instruction Multiple Data) format where there are 8, 16, 32 or 64 coordinate tuples to be processed (i.e. SIMD8, SIMD16 etc.) in the same manner. Some message types are restricted to SIMD8 and SIMD16 varieties and other are restricted to SIMD32 or SIMD64. See the section on Texture Coordinate Processing more details on texture coordinate requirements.

SIMD8 and SIMD16 messages are further organized into groups of 4 sets of coordinates which generally form a 2x2 “subspan” of texel locations. The spatial locality of the texel locations within a sub-span improves the performance of the sampler and allows the processing of the 4 texel locations in parallel. A SIMD8 message contains two subspans and a SIMD16 contains 4 subspans.

Sampler Data Fetches

The 3D sampler will automatically fetch required data from surfaces in system memory as needed to perform each sample operation. Fetched data may be stored in an internal cache to reduce latency for subsequent fetch operations.

The sampler calculates the address into a surface and uses RENDER_SURFACE_STATE state objects to determine the location within system memory and the format of the surface being fetched. Sampler can also receive or calculate the LOD (Level of Detail) of a surface if the surface supports multiple Mips and will fetch from the correct Mip in this case. See Texture Address Calculation sub-section for more detail on addresses and LOD calculation.

The sampler will also automatically decompress any supported compression format once data has been fetched. See the subsection Surface State for a list of supported surface formats, including compressed formats. Likewise, the sampler can linearize (inverse Gamma) sRGB formats prior to filtering.

Sampler Filtering and Processing

The sampler is capable of performing all basic filtering operations (point, bilinear, trilinear, anisotropic, cube etc.) based on the SAMPLER_STATE state object associated with the sample operation being requested.

In most cases, data returned is in the form of 32-bit or 16-bit IEEE floating-point per channel to ensure maximum precision. See Writeback Message section for a description of the format of returned data. Output Data is only returned to the requesting agent or written to a designated Render Target (RT). Sample results are never cached within the sampler or written to system memory.

Texture Coordinate Processing

The Texture Coordinate Processing function of the Sampling Engine performs any operations on the texture coordinates that are required before physical addresses of texel samples can be generated.

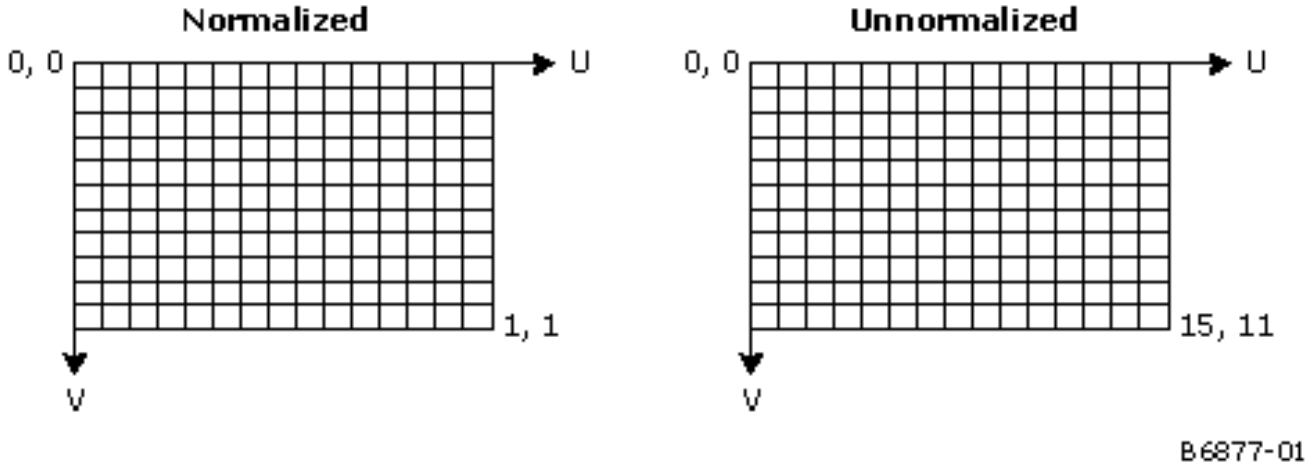
Texture Coordinate Normalization

A texture coordinate may have *normalized* or *unnormalized* values. In this function, unnormalized coordinates are normalized.

Normalized coordinates are specified in units relative to the map dimensions, where the origin is located at the upper/left edge of the upper left texel, and the value 1.0 coincides with the lower/right edge of the lower right texel . 3D rendering typically utilizes normalized coordinates.

Unnormalized coordinates are in units of texels and have not been divided (normalized) by the associated map's height or width. Here the origin is the located at the upper/left edge of the upper left texel of the base texture map.

Normalized vs. Unnormalized Texture Coordinates



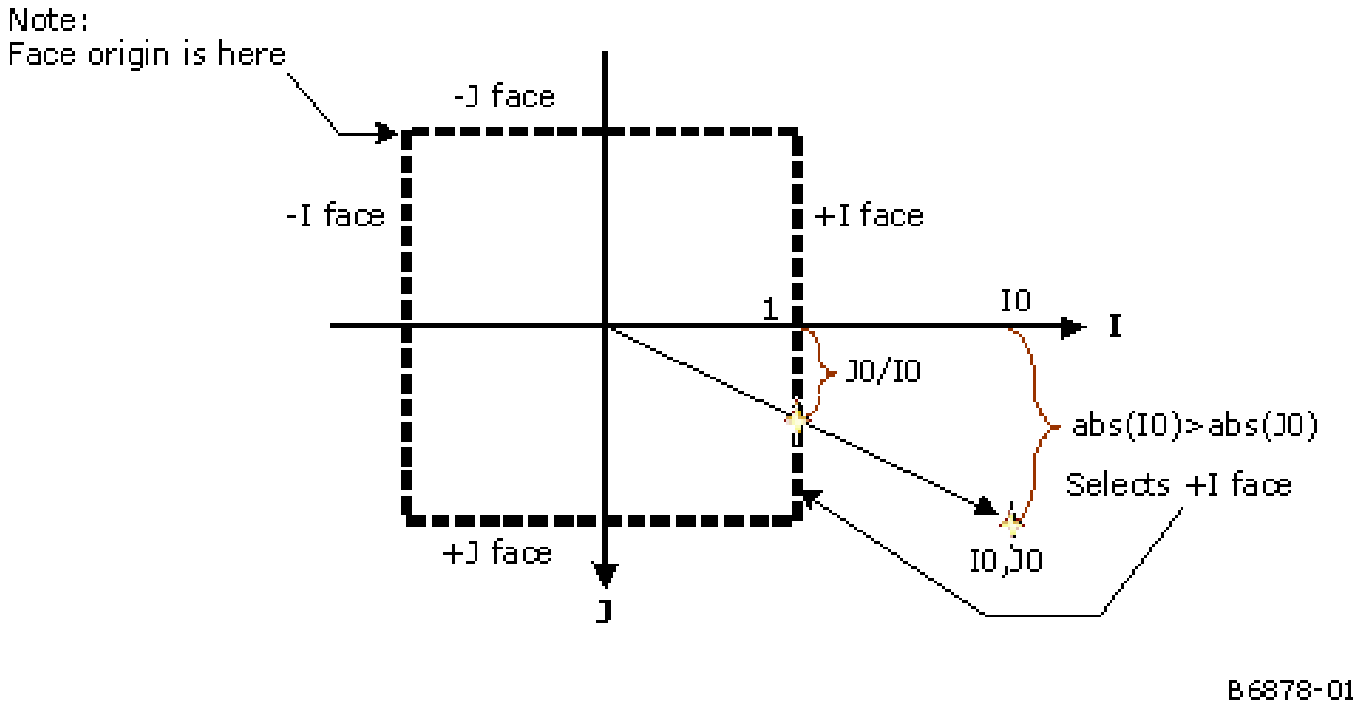
Texture Coordinate Computation

Cartesian (2D) and homogeneous (projected) texture coordinate values are projected from (interpolated) screen space back into texture coordinate space by dividing the pixel's S and T components by the Q component. This operation is done prior to sending sample operations to the 3D sampler.

Vector (cube map) texture coordinates are generated by first determining which of the 6 cube map faces (+X, +Y, +Z, -X, -Y, -Z) the vector intersects. The vector component (X, Y or Z) with the largest absolute value determines the proper (major) axis, and then the sign of that component is used to select between the two faces associated with that axis. The coordinates along the two minor axes are then divided by the coordinate of the major axis, and scaled and translated, to obtain the 2D texture coordinate ([0,1]) within the chosen face. Note that the coordinates delivered to the sampling engine must already have been divided by the component with the largest absolute value.

An illustration of this cube map coordinate computation, simplified to only two dimensions, is provided below:

Cube Map Coordinate Computation Example



Sampler SW performance hints

This is a section describing things that SW can do to get better performance out of sampler hardware.

I,L,LA,A formats

These are expanded to RGBA formats in sampler and as result are treated as larger texel sizes in sampler and have worse performance. Most of the time it is better to use the Shader Channel Select signals in RENDER_SURFACE_STATE to emulate them using R or RG formats.

		Channel select			
Base format	New format	R	G	B	A
I*	R*	R	R	R	R
L*	R*	R	R	R	1.0
LA*	RG*	R	R	R	G
A*	R*	0.0	0.0	0.0	R

Issues with this where result can be different. The A channel for LA and A format will be 0.0 rather than 1.0.

The table below shows the mapping requires for all Luminance, Intensity and Transparency formats:

Original Format	Mapped To Format	Channel select Programming (R/G/B/A)
A16_FLOAT	R16_FLOAT	0/0/0/R
A16_UNORM	R16_UNORM	0/0/0/R
A24X8_UNORM	R24_UNORM_X8_TYPELESS	0/0/0/R
A32_FLOAT	R32_FLOAT	0/0/0/R
A32_UNORM	R32_UNORM	0/0/0/R
A8_UNORM	R8_UNORM	0/0/0/R
I16_FLOAT	R16_FLOAT	R/R/R/R
I16_UNORM	R16_UNORM	R/R/R/R
I24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/R
I32_FLOAT	R32_FLOAT	R/R/R/R
I8_SINT	R8_SINT	R/R/R/R
I8_UINT	R8_UINT	R/R/R/R
I8_UNORM	R8_UNORM	R/R/R/R
L16_FLOAT	R16_FLOAT	R/R/R/1
L16_UNORM	R16_UNORM	R/R/R/1
L16A16_FLOAT	R16G16_FLOAT	R/R/R/G
L16A16_UNORM	R16G16_UNORM	R/R/R/G
L24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/1
L32_FLOAT	R32_FLOAT	R/R/R/1
L32_UNORM	R32_UNORM	R/R/R/1
L32A32_FLOAT	R32G32_FLOAT	R/R/R/G
L8_SINT	R8_SINT	R/R/R/1
L8_UINT	R8_UINT	R/R/R/1
L8_UNORM	R8_UNORM	R/R/R/1
L8A8_SINT	R8G8_SINT	R/R/R/G
L8A8_UINT	R8G8_UINT	R/R/R/G
L8A8_UNORM	R8G8_UNORM	R/R/R/G
Y8_UNORM	R8_UNORM	R/R/R/1

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

Starting in SKL 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.

LD* Messages

NativeL1 Fill rate improvement SW requirements

Texel Address Generation

To better understand texture mapping, consider the mapping of each object (screen-space) pixel onto the textures images. In texture space, the pixel becomes some arbitrarily sized and aligned quadrilateral. Any given pixel of the object may “cover” multiple texels of the map, or only a fraction of one texel. For each pixel, the usual goal is to sample and filter the texture image in order to best represent the covered texel values, with a minimum of blurring or aliasing artifacts. Per-texture state variables are provided to allow the user to employ quality/performance/footprint tradeoffs in selecting how the particular texture is to be sampled.

The Texel Address Generation function of the Sampling Engine is responsible for determining how the texture maps are to be sampled. Outputs of this function include the number of texel to be fetched, along with the physical addresses of the samples and the filter weights to be applied to the samples after they are read. This information is computed given the incoming texture coordinate and gradient values, and the relevant state variables associated with the sampler and surface. This function also applies the texture coordinate address controls when converting the sample texture coordinates to map addresses.

Level of Detail Computation (Mipmapping)

Due to the specification and processing of texture coordinates at object vertices, and the subsequent object warping due to a perspective projection, the texture image may become *magnified* (where a texel covers more than one pixel) or *minified* (a pixel covers more than one texel) as it is mapped to an object. In the case where an object pixel is found to cover multiple texels (texture minification), merely choosing one (e.g., the texel sample nearest to the pixel’s texture coordinate) will likely result in severe aliasing artifacts.

Mipmapping and texture filtering are techniques employed to minimize the effect of undersampling these textures. With mipmapping, software provides *mipmap levels,* a series of pre-filtered texture maps of decreasing resolutions that are stored in a fixed (monolithic) format in memory. When mipmaps are provided and enabled, and an object pixel is found to cover multiple texels (e.g., when a textured object is located a significant distance from the viewer), the device will sample the mipmap level(s) offering a texel/pixel ratio as close to 1.0 as possible.

The device supports up to 14 mipmap levels per map surface, ranging from 8192 x 8192 texels to a 1 X 1 texel. Each successive level has ½ the resolution of the previous level in the U and V directions (to a minimum of 1 texel in either direction) until a 1x1 texture map is reached. The dimensions of mipmap levels need not be a power of 2.

Each mipmap level is associated with a *Level of Detail (LOD)* number. LOD is computed as the approximate, $\log_2$ measure of the ratio of texels per pixel. The highest resolution map is considered LOD 0. A larger LOD number corresponds to lower resolution mip level.

The *Sampler[]BaseMipLevel* state variable specifies the LOD value at which the minification filter vs. the magnification filter should be applied.

When the texture map is magnified (a texel covers more than one pixel), the base map (LOD 0) texture map is accessed, and the magnification mode selects between the nearest neighbor texel or bilinear interpolation of the 4 neighboring texels on the base (LOD 0) mipmap.

Base Level Of Detail (LOD)

The per-pixel LOD is computed in an implementation-dependent manner and approximates the $\log_2$ of the texel/pixel ratio at the given pixel. The computation is typically based on the differential texel-space distances associated with a one-pixel differential distance along the screen x- and y-axes. These texel-space distances are computed by evaluating neighboring pixel texture coordinates, these coordinates being in units of texels on the base MIP level (multiplied by the corresponding surface size in texels). The q coordinates represent the third dimension for 3D (volume) surfaces, this coordinate is a constant 0 for 2D surfaces.

The ideal LOD computation is included below.

$$LOD(x,y) = \log_2[\rho(x,y)]$$

where :

$$\rho(x,y) = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial q}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial q}{\partial y}\right)^2}\right\},$$

LOD Bias

A biasing offset can be applied to the computed LOD and used to artificially select a higher or lower miplevel and/or affect the weighting of the selected mipmap levels. Selecting a slightly higher mipmap level will trade off image blurring with possibly increased performance (due to better texture cache reuse). Lowering the LOD tends to sharpen the image, though at the expense of more texture aliasing artifacts.

The LOD bias is defined as sum of the *LODBias* state variable and the *pixLODBias* input from the input message (which can be non-zero only for sample_b messages). The application of LOD Bias is

unconditional, therefore these variables must both be set to zero in order to prevent any undesired biasing.

Note that, while the LOD Bias is applied prior to clamping and min/mag determination and therefore can be used to control the min-vs-mag crossover point, its use has the undesired effect of actually changing the LOD used in texture filtering.

LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable. Enabling pre-clamping matches OpenGL semantics .

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 be 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:  U4.8
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

```
    PreClamp = LODPreClampMode != PRECLAMP_NONE
    if ( PreClamp )
        if ( PRECLAMP_D3D )
            LOD = min(LOD, AdjMaxLod)
            LOD = max(LOD, AdjMinLod)
        else
            LOD = min(LOD, MaxLod)
            LOD = max(LOD, MinLod)
```

```
MagMode = (LOD - Base <= 0)
```

Pseudocode

```
MagClampMipNone = LODClampMagnificationMode == MAG_CLAMP_MIPNONE
```

```
if ( (MagMode && MagClampMipNone) or MipFlt == None )
    LOD = 0
    LOD = min(LOD, ceil(AdjMaxLod))
    LOD = max(LOD, floor(AdjMinLod))
else if ( MipFlt == Nearest )
```

Pseudocode

```
LOD = min(LOD, ceil(AdjMaxLod))
LOD = max(LOD, floor(AdjMinLod))
```

```
    LOD += 0.5
    LOD = floor(LOD)
else
    // MipFlt = Linear
    LOD = min(LOD, AdjMaxLod)
    LOD = max(LOD, AdjMinLod)
    TriBeta = frac(LOD)
    LOD0 = floor(LOD)
    LOD1 = LOD0 + 1

if ( ! lod )  // "LOD" message type
    Lod += SurfMinLod
```

If Out_of_Bounds is true, LOD is set to zero and instead of sampling the surface the texels are replaced with zero in all channels, except for surface formats that don't contain alpha, for which the alpha channel is replaced with one. These texels then proceed through the rest of the pipeline.

Intra-Level Filtering Setup

Depending on whether the texture is being minified or magnified, the *MinFilter* or *MagFilter* state variable (respectively) is used to select the sampling filter to be used within a mip level (intra-level, as opposed to any inter-level filter). Note that for volume maps, this selection also applies to filtering between layers.

The processing at this stage is restricted to the selection of the filter type, computation of the number and texture map coordinates of the texture samples, and the computation of any required filter parameters. The filtering of the samples occurs later on in the Sampling Engine function.

The following table summarizes the intra-level filtering modes.

Sampler[]Min/MagFilter value	Description
MAPFILTER_NEAREST	Supported on all surface types. The texel nearest to the pixel's U,V,Q coordinate is read and output from the filter.
MAPFILTER_LINEAR	Not supported on buffer surfaces. The 2, 4, or 8 texels (depending on 1D, 2D/CUBE, or 3D surface, respectively) surrounding the pixel's U,V,Q coordinate are read and a linear filter is applied to produce a single filtered texel value.
MAPFILTER_ANISOTROPIC	Not supported on buffer or 3D surfaces. A projection of the pixel onto the texture map is generated and "subpixel" samples are taken along the major axis of the projection (center axis of the longer dimension). The outermost subpixels are weighted according to closeness to the edge of the projection, inner subpixels are weighted equally. Each subpixel samples a bilinear 2x2 of texels and the results are blended according to weights to produce a filtered texel value.
MAPFILTER_MONO	Supported only on 2D surfaces. This filter is only supported with the monochrome (MONO8) surface format. The monochrome texel block of the specified size surrounding the pixel is selected and filtered.

MAPFILTER_NEAREST

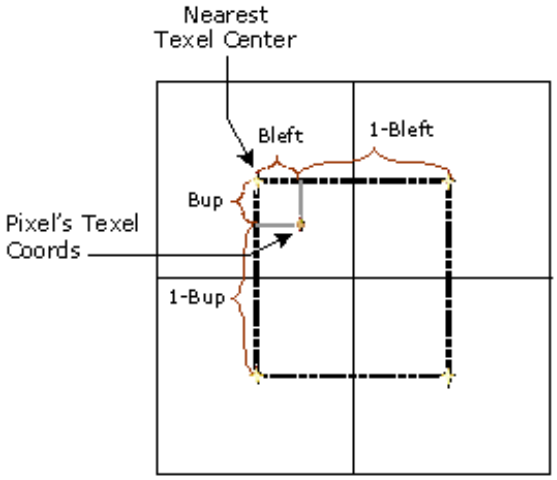
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level. This is a form of "Point Sampling".

MAPFILTER_LINEAR

The following description indicates behavior of the MIPFILTER_LINEAR filter for 2D and CUBE surfaces. 1D and 3D surfaces follow a similar method but with a different number of dimensions available.

When the MAPFILTER_LINEAR filter is selected on a 2D surface, the 2x2 region of texels surrounding the pixel's texture coordinate are sampled and later bilinearly filtered. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

Bilinear Filter Sampling

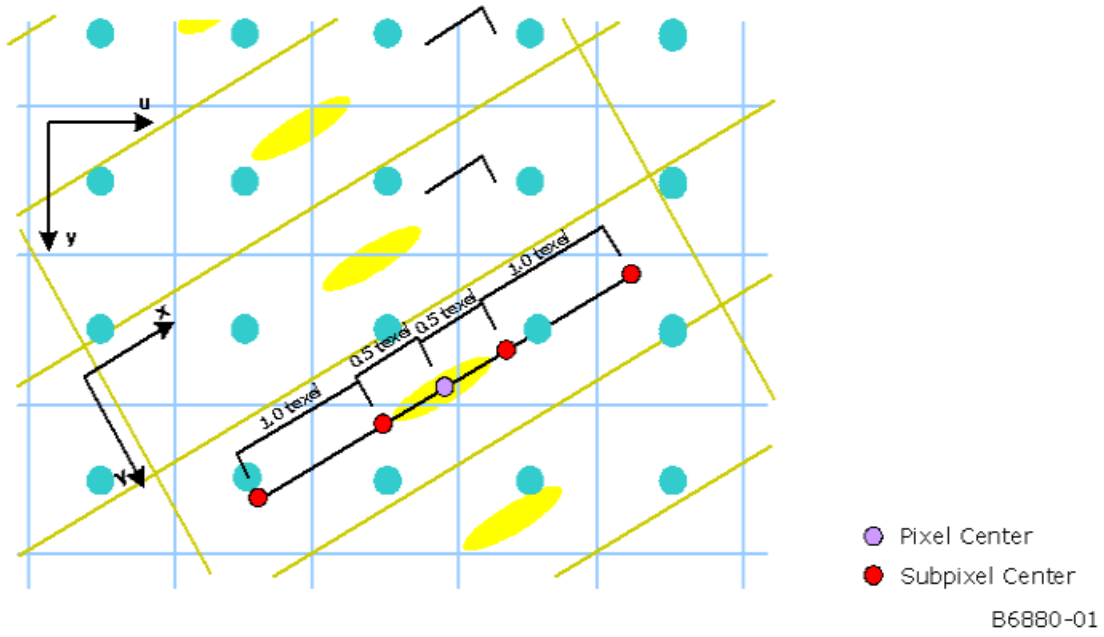


B6879-01

MAPFILTER_ANISOTROPIC

The MAPFILTER_ANISOTROPIC texture filter attempts to compensate for the anisotropic mapping of pixels into texture map space. A possibly non-square set of texel sample locations will be sampled and later filtered. The *MaxAnisotropy* state variable is used to select the maximum aspect ratio of the filter employed, up to 16:1.

The algorithm employed first computes the major and minor axes of the pixel projection onto the texture map. LOD is chosen based on the minor axis length in texel space. The anisotropic "ratio" is equal to the ratio between the major axis length and the minor axis length. The next larger even integer above the ratio determines the anisotropic number of "ways", which determines how many subpixels are chosen. A line along the major axis is determined, and "subpixels" are chosen along this line, spaced one texel apart, as shown in the diagram below. In this diagram, the texels are shown in light blue, and the pixels are in yellow.



Each subpixel samples a bilinear 2x2 around it just as if it was a single pixel. The result of each subpixel is then blended together using equal weights on all interior subpixels (not including the two endpoint subpixels). The endpoint subpixels have lesser weight, the value of which depends on how close the "ratio" is to the number of "ways". This is done to ensure continuous behavior in animation.

MAPFILTER_MONO

When the MAPFILTER_MONO filter is selected, a block of monochrome texels surrounding the pixel sample location are read and filtered using the kernel described below. The size of this block is controlled by **Monochrome Filter Height** and **Width** (referred to here as N_v and N_u, respectively) state. Filters from 1x1 to 7x7 are supported (not necessarily square).

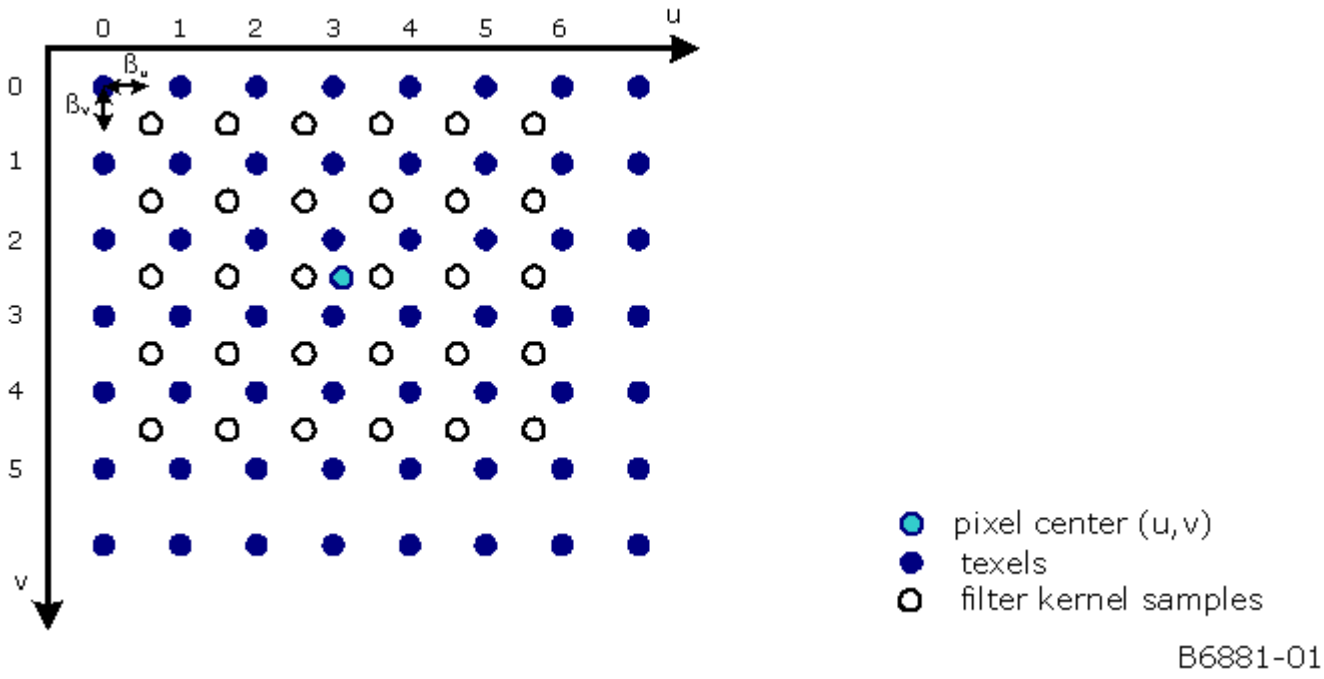
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f, v_f) relative to the pixel center at (u, v) is given by the following:

$$u_f = u - \frac{N_u}{2}$$

$$v_f = v - \frac{N_v}{2}$$

b_u and b_v are the fractional parts of u_f and v_f, respectively. The integer parts select the upper left texel for the kernel filter, given here as $T_{0,0}$.

Sampling Using MAPFILTER_MONO



The formula for the final filter output F is given by the following. Since this is a monochrome filter, each texel value (T) is a single bit, and the output F is an intensity value that is replicated across the color and alpha channels.

$$S = \frac{1}{N_u * N_v}$$

$$F = \left[(1-\beta_u)(1-\beta_v)\sum_{i=0}^{N_u-1}\sum_{j=0}^{N_v-1} T_{i,j} + \beta_u(1-\beta_v)\sum_{i=1}^{N_u}\sum_{j=0}^{N_v-1} T_{i,j} + (1-\beta_u)\beta_v\sum_{i=0}^{N_u-1}\sum_{j=1}^{N_v} T_{i,j} + \beta_u\beta_v\sum_{i=1}^{N_u}\sum_{j=1}^{N_v} T_{i,j} \right] * S$$

Inter-Level Filtering Setup

The *MipFilter* state variable determines if and how texture mip maps are to be used and combined. The following table describes the various mip filter modes:

MipFilter Value	Description
MIPFILTER_NONE	Mipmapping is DISABLED. Apply a single filter on the highest resolution map available (after LOD clamping).
MIPFILTER_NEAREST	Choose the nearest mipmap level and apply a single filter to it. Here the biased LOD will be rounded to the nearest integer to obtain the desired miplevel. LOD Clamping may further restrict this miplevel selection.
MIPFILTER_LINEAR	Apply a filter on the two closest mip levels and linear blend the results using the distance between the computed LOD and the level LODs as the blend factor. Again, LOD Clamping may further restrict the selection of miplevels (and the blend factor between them).

When minifying and MIPFILTER_NEAREST is selected, the computed LOD is rounded to the nearest mip level.

When minifying and MIPFILTER_LINEAR is selected, the fractional bits of the computed LOD are used to generate an inter-level blend factor. The LOD is then truncated. The mip level selected by the truncated LOD, and the next higher (lower resolution) mip level are determined.

Regardless of *MipFilter* and the min/mag determination, all computed LOD values (two for MIPFILTER_LINEAR, otherwise one) are then unconditionally clamped to the range specified by the (integer bits of) *MinLOD* and *MaxLOD* state variables.

Texture Address Control

The *[TCX,TCY,TCZ]ControlMode* state variables control the access and/or generation of texel data when the specific texture coordinate component falls *outside* of the normalized texture map coordinate range [0,1).

The table below provides all the supported Address Control modes for each direction.

TC[X,Y,Z] Control	Operation
TEXCOORDMODE_CLAMP	Clamp to the texel value at the edge of the map.
TEXCOORDMODE_CLAMP_BORDER	Use the texture map's border color for any texel samples falling outside the map. The border color is specified via a pointer in SAMPLER_STATE.
TEXCOORDMODE_HALF_BORDER	Similar to CLAMP_BORDER except texels outside of the map are clamped to a value halfway between the edge texel and the border color.
TEXCOORDMODE_WRAP	Upon crossing an edge of the map, repeat at the other side of the map in the same dimension.
TEXCOORDMODE_CUBE	Only used for cube maps. Here texels from adjacent cube faces can be sampled along the edges of faces. This is considered the highest quality mode for cube environment maps.
TEXCOORDMODE_MIRROR	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.

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

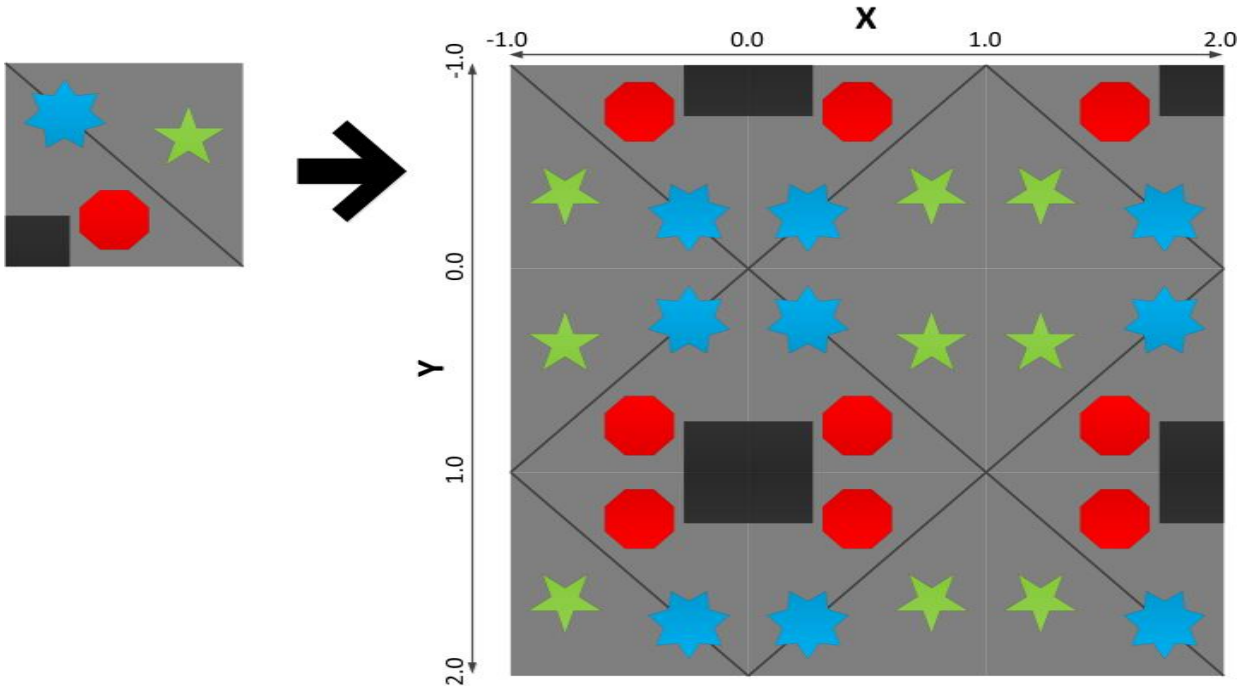
The number of map wraps on a given object is limited to 32. Going beyond this limit is legal, but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the 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.

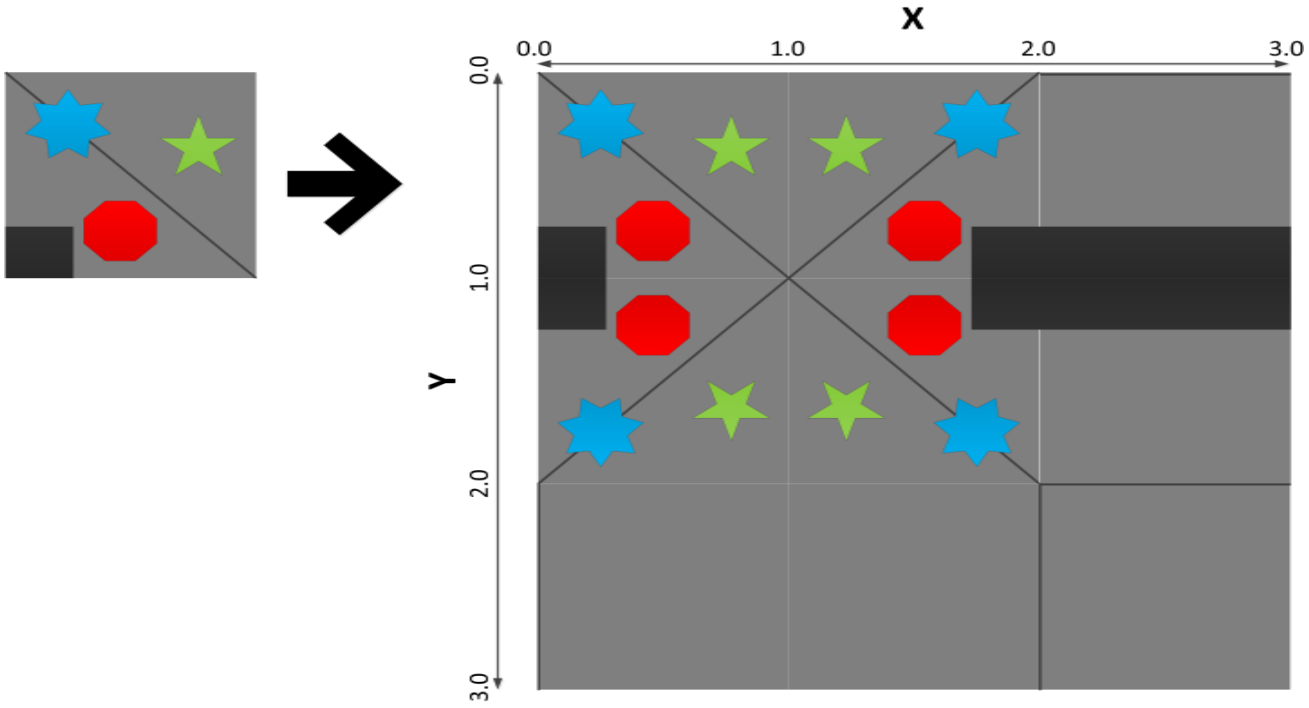
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR for both TCX and TCY is mapped.



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.

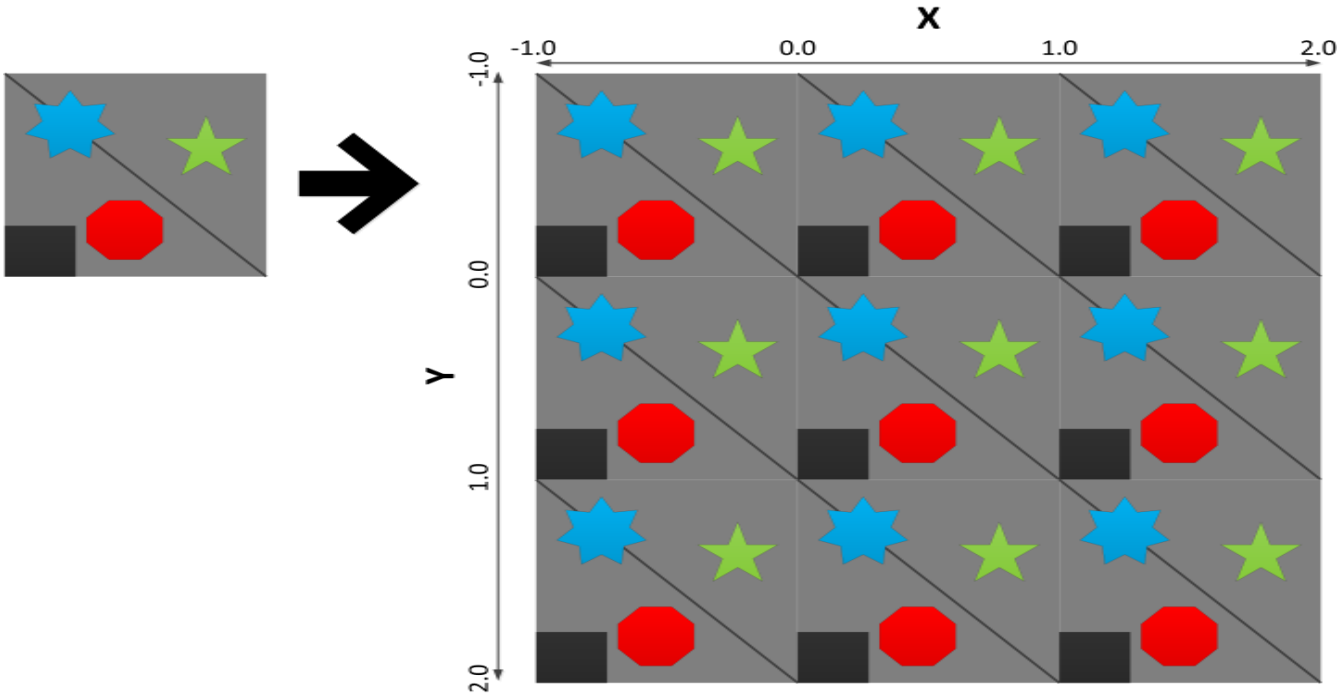
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR_ONCE for both TCX and TCY is mapped.



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.

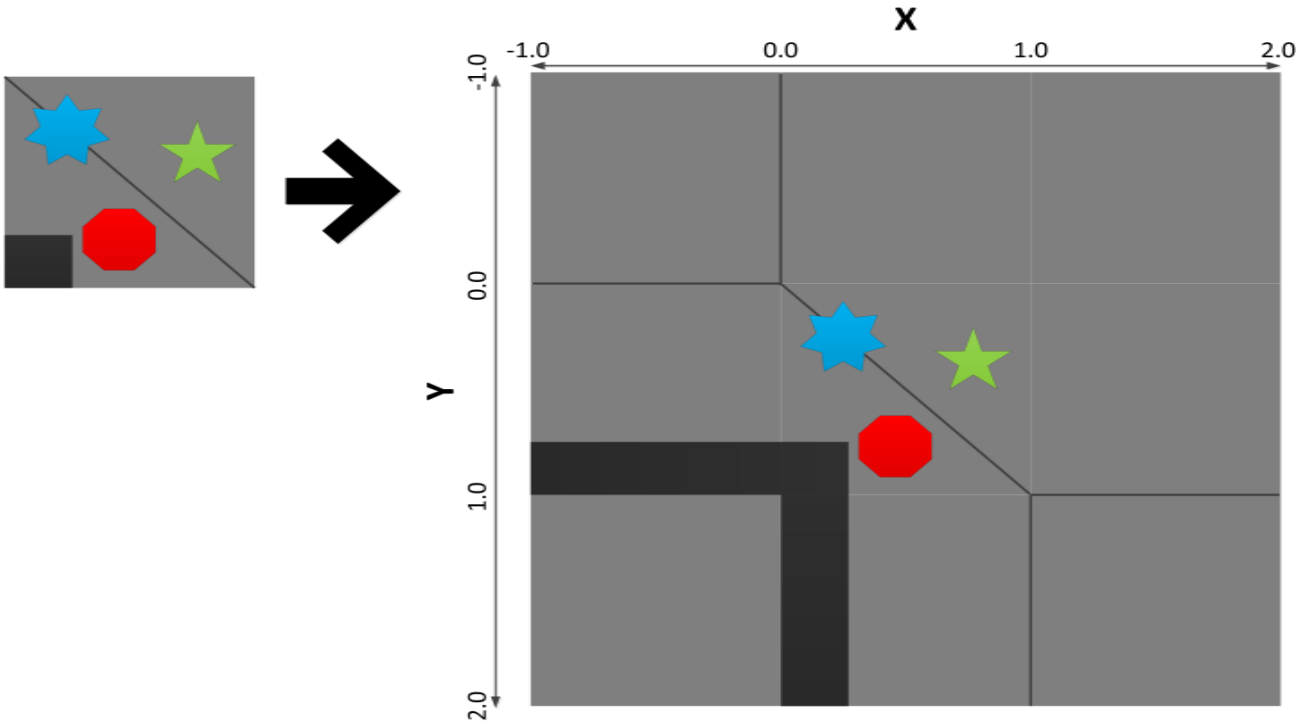


TEXCOORDMODE_CLAMP Mode

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.

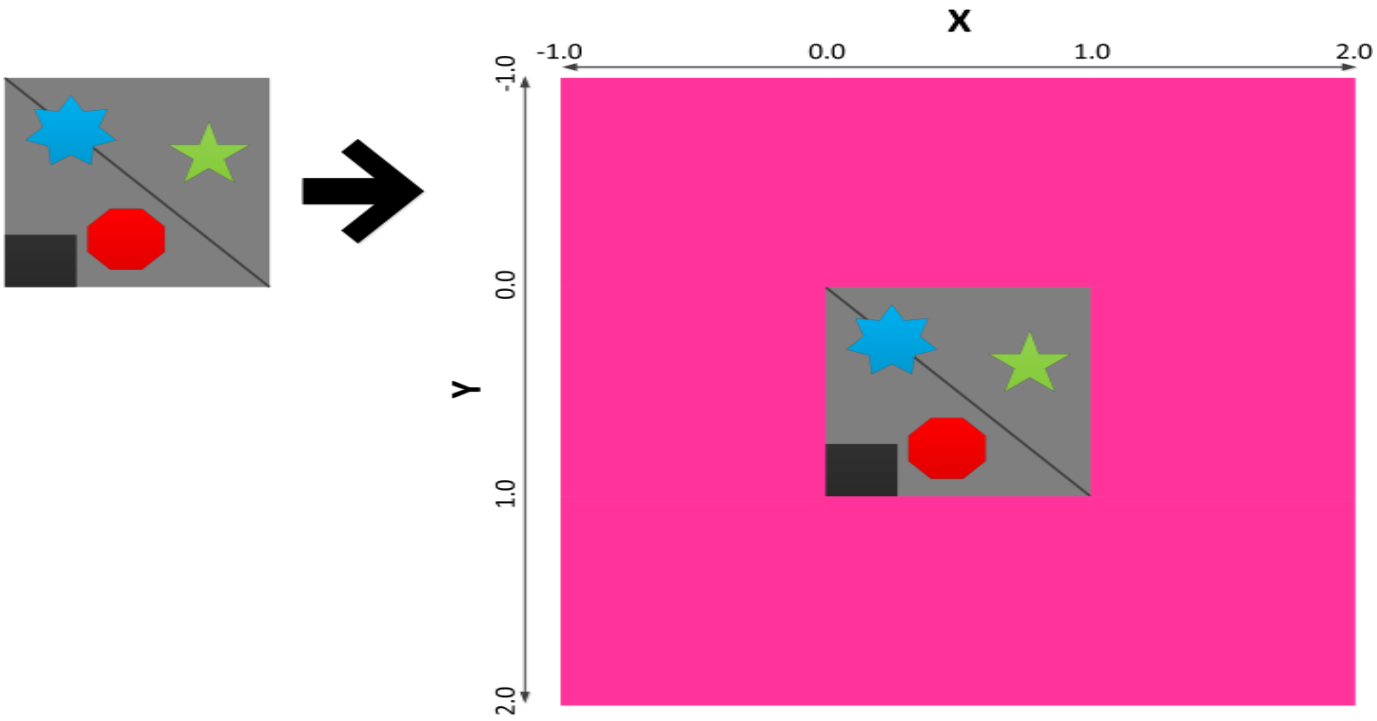
Texture Clamp Mode



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.

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 includiong "Px"), the palette lookup of the index determines the appropriate RGB values.

Texel Chroma Keying

ChromaKey is a term used to describe a method of effectively removing or replacing a specific range of texel values from a map that is applied to a primitive, e.g., in order to define transparent regions in an RGB map. The Texel Chroma Keying function of the Sampling Engine pipeline conditionally tests texel samples against a "key" range, and takes certain actions if any texel samples are found to match the key.

Chroma Key Testing

ChromaKey refers to testing the texel sample components to see if they fall within a range of texel values, as defined by *ChromaKey[][High,Low]* state variables. If each component of a texel sample is found to lie within the respective (inclusive) range and ChromaKey is enabled, then an action will be taken to remove this contribution to the resulting texel stream output. Comparison is done separately on each of the channels and only if all 4 channels are within range the texel will be eliminated.

The Chroma Keying function is enabled on a per-sampler basis by the *ChromaKeyEnable* state variable.

The *ChromaKey[][High,Low]* state variables define the tested color range for a particular texture map.

Chroma Key Effects

There are two operations that can be performed to "remove" matching texel samples from the image. The *ChromaKeyEnable* state variable must first enable the chroma key function. The *ChromaKeyMode* state variable then specifies which operation to perform on a per-sampler basis.

The *ChromaKeyMode* state variable has the following two possible values:

KEYFILTER_KILL_ON_ANY_MATCH: Kill the pixel if any contributing texel sample matches the key.

KEYFILTER_REPLACE_BLACK: Here the sample is replaced with (0,0,0,0).

The Kill Pixel operation has an effect on a pixel only if the associated sampler is referenced by a sample instruction in the pixel shader program. If the sampler is not referenced, the chroma key compare is not done and pixels cannot be killed based on it.

Shadow Prefilter Compare

When a *sample_c* message type is processed, a special shadow-mapping precomparison is performed on the texture sample values prior to filtering. Specifically, each texture sample value is compared to the "ref" component of the input message, using a compare function selected by *ShadowFunction*, and described in the table below. Note that only single-channel texel formats are supported for shadow mapping, and so there is no specific color channel on which the comparison occurs.

ShadowFunction	Result
PREFILTEROP_ALWAYS	0.0
PREFILTEROP_NEVER	1.0
PREFILTEROP_LESS	(texel < ref) ? 0.0 : 1.0
PREFILTEROP_EQUAL	(texel == ref) ? 0.0 : 1.0
PREFILTEROP_LEQUAL	(texel <= ref) ? 0.0 : 1.0
PREFILTEROP_GREATER	(texel > ref) ? 0.0 : 1.0
PREFILTEROP_NOTEQUAL	(texel != ref) ? 0.0 : 1.0
PREFILTEROP_GEQUAL	(texel >= ref) ? 0.0 : 1.0

The binary result of each comparison is fed into the subsequent texture filter operation (in place of the texel's value which would normally be used).

Software is responsible for programming the "ref" component of the input message such that it approximates the same distance metric programmed in the texture map (e.g., distance from a specific light to the object pixel). In this way, the comparison function can be used to generate "in shadow" status for each texture sample, and the filtering operation can be used to provide soft shadow edges.

Texel Filtering

The Texel Filtering function of the Sampling Engine performs any required filtering of multiple texel values on and possibly between texture map layers and levels. The output of this function is a single texel color value.

The state variables *MinFilter*, *MagFilter*, and *MipFilter* are used to control the filtering of texel values. The *MipFilter* state variable specifies how many mipmap levels are included in the filter, and how the results of any filtering on these separate levels are combined to produce a final texel color. The *MinFilter* and *MagFilter* state variables specify how texel samples are filtered within a level.

Texel Color Gamma Linearization

This function is supported to allow pre-gamma-corrected texel RGB (not A) colors to be mapped back into linear (gamma=1.0) gamma space prior to (possible) blending with, and writing to the Color Buffer. This permits higher quality image blending by performing the blending on colors in linear gamma space.

This function is enabled on a per-texture basis by use of a surface format with "_SRGB" in its name. If enabled, the pre-filtered texel RGB color to be converted to gamma=1.0 space by applying a ^(2.4) exponential function.

Multisampled Surface Behavior

Multisampled surfaces are sampled using a dedicated point-sample message **ld2dms** or **l2dms_w**. These message contain Si (sample index) which is used to sample from a specific subsample plane. Each texel of a multi-sampled surface has 1,2,4,8 or 16 "subsamples" per texel.

The sampleinfo message returns specific parameters associated with a multisample surface such as the number of subsamples per texel. The resinfo message returns the height, width, depth (in units of *pixels*, not samples), and MIP count of the surface .

Multisampled surfaces typically have an associated MCS control surface which contains information about which indicates which sample index contains the color data for a specific subample. The control surface can be accessed using the **ld_mcs** message, which returns the information for all subsamples for a given texel.

From an optimization perspective there are two different approaches which can be used to resolver a multisampled surface:

A shader can fetch ld_mcs to determine which subsamples need to be fetched using ld2dms or ld2dms_w. In general, it is expected that most subsamples will reference Sample index zero for it's color by default. It may be optimal for the shader to always fetch to sample index 0 unconditionally, and only fetch to other sample indices based on the fetched MCS. Resolve can then be done with the minimumal 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 passsed 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.

Quilted Textures]

Quilted textures allow very large 2D textures, up to 512K x 512K texels. Rather than simply increasing the height and width ranges, a quilted texture is stored in memory as a 2D array. (Refer to the GPU Overview volume for details on 2D array storage.) Each array slice has a maximum size of 16K x 16K texels, and these slices are referred to as “quilt slices.” Two fields in SURFACE_STATE, Quilt Width and Quilt Height, define the size of the quilt in units of quilt slices. A surface is defined as a quilted texture if either one of the above fields are set to a value other than 1. Quilted textures are supported with all address control modes, and with all sample* messages that support 2D arrays and use the SURFACE_STATE “for most messages.” In general, whether the surface is quilted or not is orthogonal to the message definition.

MIP maps are supported with quilted textures, with the quilt slices being reduced by a factor of 2 in each dimension for each increment in LOD. The minimum size MIP is where the quilt slice is a 1x1. Effective QuiltWidth and QuiltHeight are not reduced with increasing LOD which would cause the smallest mip to be QuiltWidth x QuiltHeight rather than the typical 1x1.

Refer to the Quilted Textures section of Common Surface Formats, Surface Layout and Tiling for details on how quilted textures are mapped into the 2D array storage layout.

The “u” and “v” parameters must be normalized, interpreted as coordinates across the entire quiltW, thus software doesn’t determine which quilt slices need to be accessed.

A new set of messages, called “SIMD8D”, is introduced to support double precision “u” and “v” parameters. The use of SIMD8D, while motivated by quilted textures, is orthogonal to them from the architecture standpoint. Both quilted and non-quilted textures can be accessed using single precision or double precision “u”/”v” parameters. Hardware will use the precision delivered to access the texture, with artifacts resulting if insufficient “u”/”v” precision is used.ves farther away from the base location more bits of beta will be lost.

Quilted textures can only be supported on tiled surfaces (e.g. TileY, TileYs, TileYf).

State

State
SW Generated BINDING_TABLE_STATE
HW Generated BINDING_TABLE_STATE

For **SAMPLER_STATE** for **Sample_8X8** see 3D-Media-GPGPU Engine > Shared Functions > Media Sampler > Sample_8x8 State > SAMPLER_STATE

The 3D sampler uses both surface state objects (RENDER_SURFACE_STATE) as well as sampler state objects (SAMPLER_STATE). These objects are cached locally in the sampler state cache

for improved performance as it is assumed that many sampler messages will utilize the same surface and sampler states.

Programming Note	
Context:	Out of Bounds Handling
If a pointer to sampler or surface state goes beyond the end of the sampler or surface state buffer (as defined by the associated size field of the STATE_BASE_ADDRESS command) the sampler will force the address offset to cache-line 0 from the defined Base Address. The result of this state fetch is undefined and depends on how the state buffer has been populated.	

Surface State Fetch

Surface state is fetched from system memory using a Binding Table Pointer (**BTP**). The **BTP** is a 16-bit value provided by the command stream (not directly by the shader) which determines the binding-table to be used. An 8-bit Binding Table Index (**BTI**) is then provided by the shader via the message descriptor, which indicates the offset into the Binding Table. The BTP and BTI are relative to the **Surface State Base Address** and the binding table itself resides in system memory. The contents of the Binding Table is a list of pointers to surface state objects. The pointer from the Binding Table is also relative to the **Sampler State Base Address,** and points directly to a 256-bit **RENDER_SURFACE_STATE** object which sampler will fetch and store in its internal state cache.

Bindless Surface State Fetch

The sampler supports a "Bindless" surface model. Bindless refers to the fact that a Binding table is not required for this mode. Instead, the shader sets the BTI to 253 in the descriptor to indicate Bindless and provides a 20-bit **Surface State Offset (SSO)** from the **Bindless Surface State Base Address .** This SSO directly selects a specific surface state object which is cached by the 3D Sampler in local State Cache. Both bindless and non-bindless (legacy) modes can be operated at the same time by the sampler.

Sampler State Fetch

SAMPLER_STATE objects are fetched independely 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_Surface_State_Base_Address** rather than the Dynamic_State_Base_Addr. Bindless Samplers can only be used in conjunction with Bindless Surfaces. The Sampler Index (SI) in the message descriptor is not used, and can be set to 0. The SAMPLER_STATE object is cached locally in the 3D sampler.

State Caching

As mentioned above, the 3D Sampler allows for automatic caching of **RENDER_SURFACE_STATE** objects and **SAMPLER_STATE** objects to provide higher performance. Coherency with system memory in the state cache, like the texture cache is handled partially by software. It is expected that the command stream or shader will issue Cache Flush operation or Cache_Flush sampler message to ensure that the L1 cache remains coherent with system memory.

Programming Note	
Context:	State Cache Coherency
Whenever the value of the Dynamic_State_Base_Addr, Surface_State_Base_Addr are altered, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the RENDER_SURFACE_STATE object in memory pointed to by the **Binding Table Pointer (BTP)** and **Binding Table Index (BTI)** is modified or SAMPLER_STATE object pointed to by the **Sampler State Pointer (SSP)** and **Sampler Index (SI)** is modified, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

Programming Note	
Context:	Bindless Surface State Cache Coherency
Whenever the value of the Bindless_Surface_State_Base_Addr is altered, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the RENDER_SURFACE_STATE object in memory pointed to by the **Surface State Offset (SSO)** is modified, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

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 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 [All]

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

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)
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
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	ASTC_LDR_2D_8x5_FLT16	3.20

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)
					001		
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
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
140h	101	0	0		101 000 000	ASTC_FULL_2D_4x4_FLT16	8.00
148h	101	1	0		101 001 000	ASTC_FULL_2D_5x4_FLT16	6.40
149h	101	1	1		101 001 001	ASTC_FULL_2D_5x5_FLT16	5.12
151h	101	2	1		101 010 001	ASTC_FULL_2D_6x5_FLT16	4.27
152h	101	2	2		101 010 010	ASTC_FULL_2D_6x6_FLT16	3.56
161h	101	4	1		101 100 001	ASTC_FULL_2D_8x5_FLT16	3.20
162h	101	4	2		101 100 010	ASTC_FULL_2D_8x6_FLT16	2.67
171h	101	6	1		101 110 001	ASTC_FULL_2D_10x5_FLT16	2.56

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)
172h	101	6	2		101 110 010	ASTC_FULL_2D_10x6_FLT16	2.13
164h	100	4	4		101 100 100	ASTC_FULL_2D_8x8_FLT16	2.00
174h	101	6	4		101 110 100	ASTC_FULL_2D_10x8_FLT16	1.60
176h	101	6	6		101 110 110	ASTC_FULL_2D_10x10_FLT16	1.28
17eh	101	7	6		101 111 110	ASTC_FULL_2D_12x10_FLT16	1.07
17fh	101	7	7		101 111 111	ASTC_FULL_2D_12x12_FLT16	0.89

Sampler Output Channel Mapping

The following table indicates the mapping of the channels from the surface to the channels output from the sampling engine. Formats with all four channels (R/G/B/A) in their name map each surface channel to the corresponding output, thus those formats are not shown in this table.

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 useage model.	

There are further restrictions listed after the table below on the use of specific surfaces.

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. See the programming notes below the following table for more details on how to support these surfaces.

Surface Format Name	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R32G32B32A32_FLOAT				R	G	B	A					
R32G32B32A32_SINT			DX10/OGL	R	G	B	A					
R32G32B32A32_UINT			DX10/OGL	R	G	B	A					
R32G32B32X32_FLOAT				R	G	B	1.0					
R32G32B32_FLOAT				R	G	B	1.0					
R32G32B32_SINT			DX10/OGL	R	G	B	1.0					
R32G32B32_UINT			DX10/OGL	R	G	B	1.0					
R16G16B16A16_SNORM				R	G	B	A					
R16G16B16A16_SINT			DX10/OGL	R	G	B	A					
R16G16B16A16_UINT			DX10/OGL	R	G	B	A					
R16G16B16A16_FLOAT				R	G	B	A					
R32G32_FLOAT			DX10/OGL	R	G	0.0	1.0		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_UNORM_SRGB				R	G	B	A					
R10G10B10A2_UINT			DX10/OGL	R	G	B	A					
R10G10B10_SNORM_A2_UNORM				R	G	B	A					
R8G8B8A8_UNORM				R	G	B	A					
R8G8B8A8_UNORM_SRGB				R	G	B	A					
R8G8B8A8_SNORM				R	G	B	A					

Surface Format Name	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R8G8B8A8_SINT			DX10/OGL	R	G	B	A					
R8G8B8A8_UINT			DX10/OGL	R	G	B	A					
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					
B10G10R10A2_UNORM_SRGB				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					
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					

Surface Format Name	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	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_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
A8P8_UNORM_PALETTE0				R	G	B	A					
A8P8_UNORM_PALETTE1				R	G	B	A					
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					
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					
P8A8_UNORM_PALETTE0				R	G	B	A					
P8A8_UNORM_PALETTE1				R	G	B	A					
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					
P4A4_UNORM_PALETTE0				R	G	B	A					
A4P4_UNORM_PALETTE0				R	G	B	A					
P8_UNORM_PALETTE0				R	G	B	A					

Surface Format Name	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
L8_UNORM_SRGB				L	L	L	1.0					
P8_UNORM_PALETTE1				R	G	B	A					
P4A4_UNORM_PALETTE1				R	G	B	A					
A4P4_UNORM_PALETTE1				R	G	B	A					
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					
P2_UNORM_PALETTE0				R	G	B	A					
P2_UNORM_PALETTE1				R	G	B	A					
BC1_UNORM		Yes		R	G	B	A					
BC2_UNORM		Yes		R	G	B	A					
BC3_UNORM		Yes		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					
MONO8				N/A	N/A	N/A	N/A					
YCRCB_SWAPUV				Cr	Y	Cb	1.0					
YCRCB_SWAPY				Cr	Y	Cb	1.0					
DXT1_RGB				R	G	B	1.0					
FXT1				R	G	B	A					
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					
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*					

Surface Format Name	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
PLANAR_420_8				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:	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:	SURFACE_STATE/Shader channel select
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:	SURFACE_STATE/Shader channel select
The BC2_NORM, BC3_UNORM, BC5_UNORM, BC5_SNORM and BC7_UNORM surface types must only be used when the **Sampler L2 Bypass Mode Disable** field in the RENDER_SURFACE_STATE is set.	

Programming Note	
Context:	EAC_SIGNED_R11 and EAC_SIGNED_RG11
When sampling on a EAC_SIGNED_R11 or EAC_SIGNED_RG11 , the returned value may be clamped to -1.0009765625 instead of -1.0	

Programming Note	
Context:	Anisotropic Filtering with 128bpt and Gen9 Border Color Mode
Anisotropic Filtering of 128bpt surfaces, when Sampler State **Texture Border Color Mode** is set to DX9, is not supported	

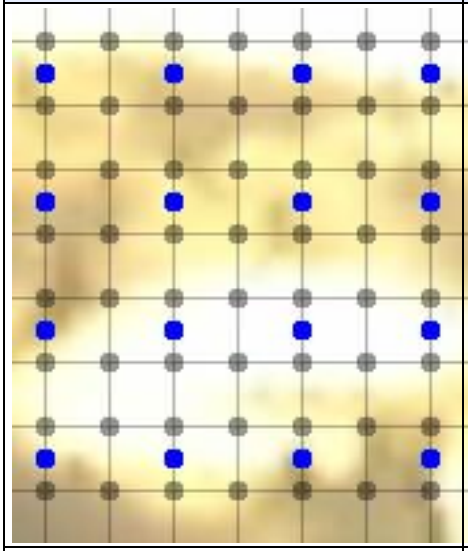
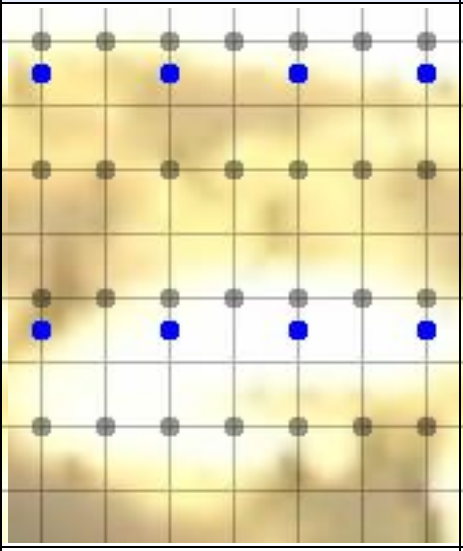
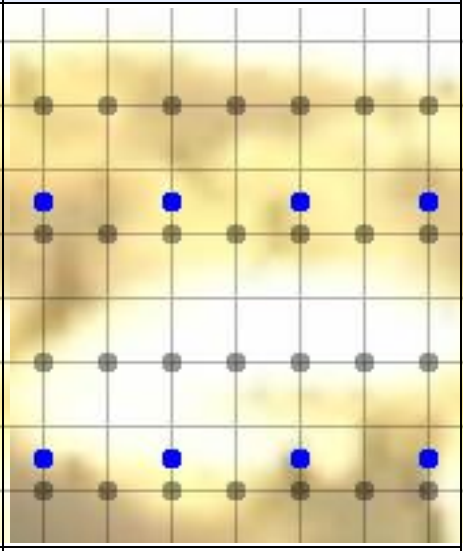
Programming Note	
Context:	NULL Surfaces and Shader Channel Select
Is SURFTYPE_NULL is selected, **Shader Channel Select Alpha** must be programmed to SCS_ZERO.	

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.

Full Frame	Top Field	Bottom Field
		
V Offset 0.5	V Offset 0.25	V Offset 0.75

MEDIA_SURFACE_STATE

Programming Note	
Context:	SURFACE_STATE for Deinterlace sample_8x
The Faulting modes described in the MEMORY_OBJECT_CONTROL_STATE should be set to the same for the multi-surface Video Analytics functions like “LBP Correlation” and “Correlation Search” for both the surfaces.	

SAMPLER_STATE

SAMPLER_STATE has different formats, depending on the message type used. For SNB+, 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.

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** Format = SNORM16
10	31:16	**Border Color Alpha** Format = SNORM16

DWord	Bits	Description
	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 R8G8B8A8_SINT R8G8_SINT

Integer Surface Format – Different Types	Surface formats
	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 be to 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
3DSTATE_SAMPLER_PALETTE_LOAD0
3DSTATE_MONOFILTER_SIZE
SAMPLER_INDIRECT_STATE

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 lenght 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.

Programming Notes

Programming Note	
Context:	3D Sampler Messages
Write back messages can return erroneous information in the Pixel Null Mask field for sample_c messages on a 64-bit surface format (e.g. R16G16B16A16_UNORM)	

Programming Note
A message header is required for GPGPU kernals in order to allow mid-thread pre-emption to allow save/restore mechanisms to work correctly.

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.

SIMD4x2. The low 8 bits of the execution mask is interpreted in groups of four. If any of the high 4 bits are asserted, that sample is valid. If any of the low 4 bits are asserted, that sample is valid. The **Write Channel Mask** rather than the execution mask determines which channels are written back to the GRF.

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, 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

Restrictions for Message Descriptors

Programming Note	
Context:	3D Sampler Messages
Use of any message to the Sampling Engine function with the **End of Thread** bit set in the message descriptor is not allowed.	

Message Header

The message header for the sampling engine is the same regardless of the message type. The messasge 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.

When Response length is 0 for sample_8x8 message then the data from sampler is directly written out to memory using media write message.

DWord	Bits	Description
M0.5	31:5	Reserved
M0.5	4:0	Output format. Only for Sample_8x8 message with direct HDC write: 0 YCRCB_NORMAL 1 YCRCB_SWAPUVY 2 YCRCB_SWAPUV 3 YCRCB_SWAPY 4 PLANAR_420_8 (NV12 only) 5 Y8_UNORM 6 Y16_SNORM 7-16 Reserved 17 Y32_UNORM 18 Reserved
M0.4	31:16	Destination Y Address (u16)

DWord	Bits	Description
M0.4	15:0	Destination X Address in bytes (u16) X Address should always be DWord-aligned.
M0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for *ld* and *resinfo* message types. This pointer is relative to the **Dynamic State Base Address**. Format = DynamicStateOfffset[31:5] The Sampler State Pointer does not have to be defined by the Message Header (many messages do not require a message header). The Sampler State Pointer may be delivered from the Command Streamer without the need for a Message Header.
M0.3	4:0	Ignored
M0.3	31:4	**Sampler State Pointer:** Specifies the 16-byte aligned pointer to the sampler state table. This field is ignored for *ld* and *resinfo* message types. This pointer is relative to the **Dynamic State Base Address** or **Bindless Surface State Base Address** depending on the setting of Bindless Surface Base Address Select bit in the GT_Mode Register. Format = StateOffset[31:4] The Sampler State Pointer does not have to be defined by the Message Header (many messages do not require a message header). The Sampler State Pointer may be delivered from the Command Streamer without the need for a Message Header.
M0.2	31:24	**Render Target or Destination Binding Table Index** Specifies the index into the binding table for the render target or HDC for messages with response length of zero (the binding table index for the sampler surface is in the message descriptor). Format = U8 Range = [0,255]
M0.2	23	**Pixel Null Mask Enable** Specifies whether the writeback message includes an extra phase indicating the pixel null mask. Refer to the *Writeback Message* section for details on format. This field must be disabled for sample+killpix and all SIMD32/64 messages. Format = Enable Ignored for Sample_8x8 message

DWord	Bits	Description
M0.2	22	**SIMD Mode Extension** If SIMD Mode in the message descriptor is set to SIMD8D/SIMD4x2, this field specifies which mode is used. For other SIMD Modes, this field is ignored. 0: SIMD8D 1: SIMD4x2
M0.2	21	Reserved
M0.2	20	Reserved
M0.2	19:18	**SIMD32/64 Output Format Control** Specifies the output format of SIMD32/64 messages (sample_unorm* and sample_8x8). Ignored for other message types. Refer to the writeback message formats for details on how this field affects returned data. This field is ignored for sample_8x8 messages if the Function is not AVS and MinMaxFilter. For MinMaxFilter only 16 bit Full and 8 bit Full modes are supported. This field is ignored and not used for HDC write message. 0: 16 bit Full 1: 16 bit Chrominance Downsampled 2: 8 bit Full 3: 8 bit Chrominance Downsampled This feature should be programmed to 0h because non-0 values may cause data corruption in returned values.
M0.2	17:16	**Gather4 Source Channel Select:** Selects the source channel to be sampled in the gather4* messages. Ignored for other message types. 0: Red channel 1: Green channel 2: Blue channel 3: Alpha channel For gather4*_c messages, this field must be set to 0 (Red channel).

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>M0.2</td><td>15</td><td>Alpha Write Channel Mask: Enables the alpha channel to be written back to the originating thread.
0: Alpha channel is written back.
1: Alpha channel is not written back.
<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2">A message with all four channels masked is not allowed.
This field is ignored for the deinterlace message.
This field must be set to zero for sample_8x8 in VSA mode.
For Sample_8x8 messages, Alpha/Blue/Red channels should be always masked (set to 1) and only Green channel is enabled (set to 0).
This field must be set to zero for all gather4* messages.</td></tr></table></td></tr>
<tr><td>M0.2</td><td>14</td><td>Blue Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
<tr><td>M0.2</td><td>13</td><td>Green Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
<tr><td>M0.2</td><td>12</td><td>Red Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
<tr><td>M0.2</td><td>11.8</td><td>U Offset: The u offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
This field is ignored if the offu parameter is included in the gather4* messages.</td></tr></table>
<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>Non-Normalized Floating-Point Coordinates</td></tr><tr><td colspan="2">Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.</td></tr></table></td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>M0.2</td><td>7:4</td><td>V Offset: The v offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>3DSampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
This field is ignored if the offu parameter is included in the gather4* messages.</td></tr></table>
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Non-Normalized Floating-Point Coordinates</td></tr><tr><td colspan="2">Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.</td></tr></table></td></tr>
<tr><td>M0.2</td><td>3:0</td><td>R Offset: The r offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>3D Sampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.</td></tr></table>
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Non-Normalized Floating-Point Coordinates</td></tr><tr><td colspan="2">Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.</td></tr></table></td></tr>
<tr><td>M0.1</td><td>31:0</td><td>Reserved</td></tr>
<tr><td>M0.0</td><td>31:0</td><td>Reserved</td></tr>
</table>

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

Variant Suffix	Definition
	coordinates. This is then used to calculate other pixel coodinates and the LOD
_po	Pixel Offset: Used only with gather4-type messages, it means the message provides a set of non-normalized integer texel offsets between +31 and -32 which are added to the calculated surface position.
_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.
_min	Minimum Value: Forces return of the minimum channel value for all contributing texels a filter operation.
_max	Maximum Value: Forces return of the maximum channel value for all contributing texels a filter operation.

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_*	The surface is sampled using the indicated sampler state. LOD is computed using calculated gradients between adjacent pixels. One, two, or three floating-point coordinate parameters may be specified depending on how many dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. The sample message enables filtering/blending operations (e.g. MAP_LINEAR) and addressing modes

Message Type	Description
	(e.g. Mirror) which are controlled by SAMPLER_STATE.

Supported Variants

Message	Description
sample	Basic sample operation as described above.
sample_l	Basic sample with LOD in message, not computed
sample_b	Basic sample with Bias in messaged added to computed LOD.
sample_c	Basic sample with Reference value in message and comparison against Red channel of the sampled surface returned. Used primarily for shadow maps. Comparison type is defined by the **Shadow Function** field in the SAMPLER_STATE.
sample_lz	Basic sample with LOD forced to 0. Possibly higher performance than **sample_l** for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed or sent in message.
sample_l_c	sample_c with LOD override in message, not computed.
sample_b_c	sample_c with LOD Bias
sample_c_lz	Similar to sample_c with LOD forced to 0. Possibly higher performance than **sample_c** or **sample_l_c** for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed.
sample_d	The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The *r* coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The *r* coordinate contains the faceid, and the r gradients are ignored by hardware. Previously known as sample_g.
sample_d_c	Same as sample_d, but returns comparison against Red Channel of sampled surface like sample_c. Previously known as sample_g_c.
sample_killpix	Basic sample, but returns a Kill Pixel Mask based on Chroma Key sampler comparison results. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask.
sample_min	Basic sample, but returns minimum value of all contributing texels for each individual color channel.
sample_max	Basic sample, but returns maximum value of all contributing texels for each individual color channel.

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.

Programming Note
The Surface Format of the associated surface cannot be MONO8. Sample is not supported in SIMD4x2 mode.
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. The Surface Format of the associated surface cannot be MONO8. 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 is not supported in SIMD4x2 mode.
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 is not supported in SIMD4x2 mode.
sample_c and sample_c_lz: Must use the DX10 BORDER_COLOR_MODE (selected in sampler state) when using CLAMP_BORDER addressing mode.
The *mlod* parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_l_c

Programming Note
sample_l_c: All restrictions applying to both sample_l and sample_c must be honored.
sample_l_c is allowed as a SIMD4x2 message.
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. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
sample_d: The mlod parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_d_c

Programming Note
sample_d_c: All restrictions applying to both sample_d and sample_c must be honored.
sample_d_c is allowed as a SIMD4x2 message.

Restrictions and Programming Notes for sample_killpix

Programming Note
sample_killpix: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. sample_killpix is supported only in SIMD8 mode. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
sample_killpix: If the sampler is disable via the **Sampler Disable** bit in the SAMPLER_STATE, the Kill Pixel Mask returned will be undefined.

Restrictions and Programming Notes for sample_min, sample_max

Programming Note
The Surface Type of the associated surface must be SURFTYPE_2D and **Surface Array** must be disabled. The Surface Format of the associated surface cannot be MONO8 Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1
If sampling on a 3D surface (Volumetric),the following two conditions cannot be supported on a sample_min or sample_max at the same time: • Sampler state addresse mode for q direction specifies D3D11_TEXADDRESS_BORDER • Address mode for U and V do not specify D3D11_TEXADDRESS_BORDER

ld Message Types

ld Message Definition

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 becasuse LOD is not sent as part of the message.

Message	Description
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	Same as ld2dms but provides more bits for MCS to support multi-sampled surfaces which are X16 MSAA. Therefore, the ld2dms_w message has two MCS parameters, named mcsl and mcsh. These parameters contain the low and high halves of the 64-bit MCS value, respectively. However, use of the ld2dms_w message is allowed with Number of Multisamples set to 2, 4, 8, or 16. With this message, if Number of Multisamples is 2, 4 or 8, the ld2dms_w message behaves the same as ld2dms, with mcsl used for mcs, and mcsh ignored. For SIMD8/SIMD16 the mcsh parameters is only used for X16 MSAA, and should be all 0's for X8 MSAA or lower depth.
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 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
ld: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1

Restrictions and Programming Notes for ld_lz

Programming Note
ld_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message. The Surface Format of the associated surface cannot be MONO8.

Restrictions and Programming Notes for ld_mcs

Programming Note

Programming Note
ld_mcs: The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.
ld_mcs: The ld_mcs message uses the Auxiliary Surface Base Address and Auxiliary Surface Pitch fields in SURFACE_STATE to determine the base address and pitch of the surface. Surface Format is overridden to R8_UINT if Number of Multisamples is 2 or 4, R32_UINT if Number of Multisamples is 8, or R32G32_UINT if Number of Multisamples is 16. This message cannot be used on a non-multisampled surface. Otherwise, ld_mcs behaves like the ld message. If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.
ld_mcs: If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_CCS_D, this message returns zeros in all channels. 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.
ld_mcs: All LOD parameters must be the same value within a single message (i.e. all samples retured for message must be from a single MIP).

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 valie data. The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface cannot be MONO8.
ld_2dms: All LOD parameters must be the same value within a single message (i.e. all samples retured 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.

Programming Note
The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface cannot be MONO8.
ld_2dms: All LOD parameters must be the same value within a single message (i.e. all samples retured for message must be from a single MIP).

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 retured for message must be from a single MIP).

gather4 Message Types

Definitions

Message Type	Description
gather4_*	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. 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: upper left sample = alpha channel / upper right sample = blue channel lower left sample = red channel / lower right sample = green channel Returned color channel is selected through a message field: **Gather4 Source Channel Select** Two or three floating-point coordinates may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters default to 0.

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

Message	Description
	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
gather4: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. 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: • 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.
gather4: If Surface Format is R10G10B10_SNORM_A2_UNORM and Gather4 Source Channel Select is alpha channel, the returned value may be incorrect.
gather4: When using gather4 type messages on CUBEMAP surfaces with SINT* surface formats the ChromaKeyEnable state bit should be set and the ChromaKeyMode state set to KEYFILTER_KILL_ON_ANY_MATCH.

Restrictions and Programming Notes for gather4_c:

Programming Note
gather4_c: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, or R16_UNORM. The channel selected by the Gather4 Source Channel Select field in the message header must be set to **Red**. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. Using DX9 Border Color Mode and either of the following is undefined:

Programming Note
• 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.
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.

Restrictions and Programming Notes for gather4_po:

Programming Note
gather4_po: The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. Using DX9 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.
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.

Restrictions and Programming Notes for gather4_po_c:

Programming Note
gather4_po_c: The Surface Type of the associated surface must be SURFTYPE_2D. 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**. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. Using DX9 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.
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.

sampleinfo Message Type

sampleinfo Message Definition

Message Type	Description
sampleinfo	The surface indicated in the surface state is not sampled. Instead, the number of samples (UINT32) and the sample position palette index (UINT32) for the surface are returned in the red and alpha channels respectively as UINT32 values. The sample position palette index returned in alpha is incremented by one from its value in the surface state. The **Sampler State Pointer** and **Sampler Index** are ignored. The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_NULL.

Supported Variants:

None

Restrictions and Programming Notes for sampleinfo:

Programming Note
sampleinfo: The return format for sampleinfo message must be 32-bits.

LOD Message Type

LOD Message Definition

Message Type	Description
LOD	The surface indicated in the surface state is not sampled. Instead, LOD is computed as if the surface will be sampled, using the indicated sampler state, and the clamped and unclamped LOD values are returned in the red and green channels, respectively, in FLOAT32 format. The blue and alpha channels are undefined, and can be masked to avoid returning them. LOD is computed using gradients between adjacent pixels. Three parameters are always specified, with extra parameters not needed for the surface being ignored.

Supported Variants:

None

Restrictions and Programming Notes for LOD:

Programming Note
LOD: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
LOD is not supported in SIMD4x2 mode.
LOD: The return format for LOD message must be 32-bits.

resinfo Message Type

resinfo Message Definition

Message Type	Description
resinfo	The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned as indicated in the table below. The format of the returned data is UINT32. In the case of a 1D or 2D surfaces, the depth value returned is the number of array slices. For non-arrayed 1D and 2D surfaces the value returned will be 1. The width, height, and depth may be scaled by the LOD parameter provided in the message to give the correct dimensions of the specified mip level. The LOD parameter is an unsigned 32-bit integer in this mode (note that negative values are out-of-range when interpreted as unsigned integers). The Sampler State Pointer and Sampler Index are always ignored. For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, if the delivered LOD is outside of the range [0..MipCount-1], the returned values in the red, green, and blue channels are 0s.

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)*	Surface Array ? Depth+1 : 0	MIP Count

Surface Type	Red	Green	Blue	Alpha
		(QuiltHeight+1)		
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_STRBUF	Buffer size (from combined Depth/Height/Width) If buffer size is exactly 2^32, zero is returned in this field.	Undefined	Undefined	Undefined
SURFTYPE_NULL	0	0	0	0

Supported Variants:

None

Restrictions and Programming Notes for resinfo:

Programming Note
resinfo: When a surface of type SURFTYPE_NULL is accessed by resinfo, the MIPCount returned is undefined instead of 0.
resinfo: The return format of resinfo message must be 32-bits.

cache_flush Message Type

cache_flush Message Definition

Message Type	Description
cache_flush	The texture caches in the sampling engine are invalidated. This includes all levels of texture cache, however the state caches are not invalidated. Any outstanding sample operations which arrive at the texture sampler prior to the cache_flush message are completed normally. Any sample operations which arrive after the cache_flush message will cause texture cache-misses and sampler will fetch textures from memory.

Supported Variants:

None

Restrictions and Programming Notes for cache_flush:

Programming Note
cache_flush: Software must ensure that this message is ordered appropriately with other messages. Hardware ensures only that this message is processed in the same order as the messages are received. The message causes a write back message to indicate that the flush has been completed. This message must be sent as a SIMD32/64 type message with header only (no additional message contents)

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_*	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: U(x,y) = U(0,0) + DeltaU * x V(x,y) = V(0,0) + DeltaV * y Typically this message and its variants are used for media applications.

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

Message	Description
	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
The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface must be UNORM with <= 8 bits per channel. The MIP Count, Depth, Surface Min LOD, Resource Min LOD, and Min Array Element of the associated surface must be 0. The Min and Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR. The Mip Mode Filter must be MIPFILTER_NONE. The TCX and TCY Address Control Mode cannot be any of: • TEXCOORDMODE_CLAMP_BORDER • TEXCOORDMODE_HALF_BORDER • TEXCOORDMODE_MIRROR_ONCE • TEXCOORDMODE_WRAP Map Width must be >= 4. Map Height must be >= 4 • Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
Map Width must be >=16 Map Height must be >=16

sample_8x8 Message Type

Supported Variants:

None.

Restrictions and Programming Notes for sample_8x8:

Programming Note
The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface must be UNORM with <= 10 bits per channel. *DeltaV * Height* of the associated surface must be less than 16.0. • *Map Width* must be >= 4.
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.

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 dfferent 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
SIMD4X2	
SIMD8	8 samples with 32-bit Coordinates
SIMD16	16 samples with 32-bit coordinates
SIMD32/64	Media Operations on 32 or 64 pixels
SIMD8D	8 pixels in 2 sub-spans with 64-bit Coordinates

Message Length

The **SIMD Mode** field determines the number of instances (i.e. pixels) to be sampled and the formatting of messages. The **Message Length** field indicates the number of parameters sent with the message. Higher-numbered parameters are optional, and default to a value of 0 if not sent but needed for the surface being sampled. The **Header Present** field determines whether a header is delivered as the first phase of the message or the default header from R0 of the thread's dispatch is used. A header will increase the length of the message by 1. Note that messages with more than 4 parameters will perform lower due to the additional information being sent to the sampler than messages with 4 or fewer parameters. For example, a sample message using 5 parameters will not be able to sustain the same throughput as a sample message with only 4 valid parameters.

The table below shows all of the SIMD modes supported by the sampling engine and how the Message Length is calculated for each. The variable "N" is the number of valid parameters, and "H" is 1 if the header is present, 0 otherwise. The maximum message length allowed to the sampler is 11. The sample_d, sample_b_c, and sample_l_c messages are not allowed with a SIMD Mode of SIMD16 because the message length is greater than 11.

Message Lengths for Supported SIMD Modes

SIMD Mode	Message Length
SIMD4X2	H+(N/4)
SIMD8	H+N
SIMD8D	H+(2*N)
SIMD16	H+(2*N)

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
SIMD4X2	32	1
SIMD8	32	4+k
SIMD8	16	2**
SIMD8D	32	4+k
SIMD8D	16	2**

SIMD Mode	Return Precision (bits per channel)	Response Length
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.

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*

** For sample_unorm, phases in the response length are reduced by 2 for each channel that is masked.

Sample8x8 Footnote

* 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
	32-bit	8	0	0	0	0

If **Pixel Fault Mask Enable** is enabled, response length must be set to a value one larger than that indicated in the tables above.

Programming Note	
Context:	3D Sampler Messages - Message Format
Parameter 0 is required.	

Message Formats

The table below describes the format of each message type for every supported SIMD type.

=The Message Type field in the message descriptor in combination with the Message Length to determine which message is being sent. The table defines all of the parameters sent for each message type. The position of the parameters in the payload is given in the section following corresponding to the *SIMD mode* given in the table. The Mnemonic column specifies the DX10-equivalent shader instructions expected to be translated to each message type.

Message parameters are typically formed in sequential 256-bit registers and sent in order to the sampler to form a complete message. All parameters are of type IEEE_Float, except those in the The ld*, resinfo, bufferinfo, 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.

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.	

Parameter Types

sample*, LOD, and gather4 messages

For all of the sample*, LOD, and gather4 message types, all parameters are 32-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.

Programming Note	
Context:	3D Sampler Messages Parameter Types
For the SIMD8D type messages, the u* and v* parameters are normalized 64-bit floating point.	

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

mcs parameter

The ‘mcs’ parameter delivers the multisample control data. The format of this parameter is always a 32-bit unsigned integer. Refer to the section titled “Multisampled Surface Behavior” for details on this parameter.

Ld* messages

For the ld message types, all parameters are 32-bit unsigned integers, 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 Payload Parameter Definition section. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2.7	31:0	**Subspan 3, Pixel 3 (lower right) Parameter 0**
M2.6	31:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**
M2.5	31:0	**Subspan 3, Pixel 1 (upper right) Parameter 0**
M2.4	31:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**

DWord	Bits	Description
M2.3	31:0	**Subspan 2, Pixel 3 (lower right) Parameter 0**
M2.2	31:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M2.1	31:0	**Subspan 2, Pixel 1 (upper right) Parameter 0**
M2.0	31:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M3 – Mn		Repeat packets 1 and 2 to cover all required parameters.

SIMD8 Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels.

DWord	Bits	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in the Payload Parameter Definition section. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 – Mn		Repeat packet 1 to cover all required parameters.

SIMD8D 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. The "u" and "v" parameters are delivered in double precision floating point, and thus it takes two message phases to deliver 8 values. These are labeled in the Payload Parameter Definition table as "u0", "u1", "v0", and "v1". The number after the u/v indicates which subspan is contained in that parameter.

Parameters "u0", "u1", "v0", or "v1"

DWord	Bits	Description
Mn.7	31:0	Pixel 3 (lower right) Parameter n – upper 32 bits Specifies the value of the pixel's parameter n. The actual parameter that maps to parameter n is given in the table in the Payload Parameter Definition section. Format = Double precision IEEE Float, upper 32 bits
Mn.6	31:0	Pixel 3 (lower right) Parameter n – lower 32 bits Format = Double precision IEEE Float, lower 32 bits
Mn.5	31:0	Pixel 2 (lower left) Parameter n – upper 32 bits
Mn.4	31:0	Pixel 2 (lower left) Parameter n – lower 32 bits
Mn.3	31:0	Pixel 1 (upper right) Parameter n – upper 32 bits
Mn.2	31:0	Pixel 1 (upper right) Parameter n – lower 32 bits
Mn.1	31:0	Pixel 0 (upper left) Parameter n – upper 32 bits
Mn.0	31:0	Pixel 0 (upper left) Parameter n – lower 32 bits

All Other Parameters

DWord	Bits	Description
Mn.7	31:0	Subspan 1, Pixel 3 (lower right) Parameter n Specifies the value of the pixel's parameter n. The actual parameter that maps to parameter n is given in the table in the Payload Parameter Definition section. Format = IEEE Float
Mn.6	31:0	Subspan 1, Pixel 2 (lower left) Parameter n
Mn.5	31:0	Subspan 1, Pixel 1 (upper right) Parameter n
Mn.4	31:0	Subspan 1, Pixel 0 (upper left) Parameter n
Mn.3	31:0	Subspan 0, Pixel 3 (lower right) Parameter n
Mn.2	31:0	Subspan 0, Pixel 2 (lower left) Parameter n
Mn.1	31:0	Subspan 0, Pixel 1 (upper right) Parameter n
Mn.0	31:0	Subspan 0, Pixel 0 (upper left) Parameter n

SIMD4x2 Payload

DWord	Bits	Description
M1.7	31:0	**Sample 1 Parameter 3** Specifies the value of the pixel's parameter 3. The actual parameter that maps to parameter 3 is given in the table in the Payload Parameter Definition section. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Sample 1 Parameter 2**
M1.5	31:0	**Sample 1 Parameter 1**
M1.4	31:0	**Sample 1 Parameter 0**
M1.3	31:0	**Sample 0 Parameter 3**
M1.2	31:0	**Sample 0 Parameter 2**
M1.1	31:0	**Sample 0 Parameter 1**
M1.0	31:0	**Sample 0 Parameter 0**
M2		Parameters 4-7 if present
M3		Parameters 8-11 if present

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 incorect 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.	

SIMD16

Return Format = 32-bit

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the

writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3. The pixels written within each destination register is determined by the execution mask on the “send” instruction.

: If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel’s red channel. Format = IEEE Float, S31 signed 2’s comp integer, or U32 unsigned integer.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1.7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
W1.6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
W1.5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
W1.4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
W1.3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
W1.2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
W1.1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
W1.0	31:0	**Supspan 2, Pixel 0 (upper left) Red**
W2		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations

DWord	Bit	Description
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** is enabled.
W8.0	31:16	Reserved: always written as 0xffff
W8.0	15:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel.

Return Format = 16-bit

:A SIMD16 writeback message with **Return Format** of 16-bit consists of up to 4 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the "send" instruction.

: If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bit	Description
W0.7	31:16	**Subspan 3, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Half Float, S15 signed 2's comp integer, or U16 unsigned integer.
	15:0	**Subspan 3, Pixel 2 (lower left) Red**
W0.6	31:16	**Subspan 3, Pixel 1 (upper right) Red**
	15:0	**Supspan 3, Pixel 0 (upper left) Red**
W0.5	31:16	**Subspan 2, Pixel 3 (lower right) Red**
	15:0	**Subspan 2, Pixel 2 (lower left) Red**
W0.4	31:16	**Subspan 2, Pixel 1 (upper right) Red**
	15:0	**Supspan 2, Pixel 0 (upper left) Red**
W0.3	31:16	**Subspan 1, Pixel 3 (lower right) Red**

	15:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.2	31:16	**Subspan 1, Pixel 1 (upper right) Red**
	15:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.1	31:16	**Subspan 0, Pixel 3 (lower right) Red**
	15:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.0	31:16	**Subspan 0, Pixel 1 (upper right) Red**
	15:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Green:** See W0 definition for pixel locations
W2		**Blue:** See W0 definition for pixel locations
W3		**Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written): W4 is only delivered when **Pixel Fault Mask Enable** is enabled.
W4.0	31:16	Reserved: always written as 0xffff
W4.0	15:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel.

SIMD8

Return Format = 32-bit

: A SIMD8* writeback message consists of up to 4 destination registers (5 in the case of sample+killpix). Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in the destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the *send* instruction.

For the sample+killpix message types, an additional register (W4) is included after the last channel register.

: If **Pixel Null Mask Enable** is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bits	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**

DWord	Bits	Description
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written) **:** This W4 is only delivered for the sample+killpix message or when **Pixel Null Mask Enable** in Message Header is set.
W4.0	31:16	**Dispatch Pixel Mask:** For his field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread for sample+killpix messages.
W4.0	15:0	**Active Pixel Mask:** For sample+killpix messages this field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. Since the SIMD8 message applies to only 8 pixels, only the low 8 bits within this field are used. The high 8 bits are always set to 1.
W4.0	31:24	Reserved: always written as 0xff when **Pixel Null Mask Enable** in Message Header is set.
W4.0	23:16	Reseved: always written as 0xff when **Pixel Null Mask Enable** in Message Header is set.
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

[CHV], :A SIMD8* writeback message with **Return Format** of 16-bit consists of up to 4 destination registers). Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in the destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the send instruction.

: If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bits	Description
W0.7:4		Reserved
W0.3	31:16	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Half Float, S15 signed 2's comp integer, or U16 unsigned integer.
	15:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.2	31:16	**Subspan 1, Pixel 1 (upper right) Red**
	15:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.1	31:16	**Subspan 0, Pixel 3 (lower right) Red**
	15:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.0	31:16	**Subspan 0, Pixel 1 (upper right) Red**
	15:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
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	Reserved: always written as 0xff
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.

SIMD4x2

A SIMD4x2 writeback message always consists of a single message register containing all four channels of each of the two "pixels" (called "samples" here, as they are not really pixels) of data. The write channel mask bits as well as the execution mask on the "send" instruction are used to determine which of the channels in the destination register are overwritten. If any of the four execution mask bits for a sample is asserted, that sample is considered to be active. The active channels in the write channel mask will be written in the destination register for that sample. If the sample is inactive (all four execution mask bits deasserted), none of the channels for that sample will be written in the destination 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	Bit	Description
W0.7	31:0	**Sample 1 Alpha:** Specifies the value of the pixel's alpha channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer.
W0.6	31:0	**Sample 1 Blue**

DWord	Bit	Description
W0.5	31:0	**Sample 1 Green**
W0.4	31:0	**Sample 1 Red**
W0.3	31:0	**Sample 0 Alpha**
W0.2	31:0	**Sample 0 Blue**
W0.1	31:0	**Sample 0 Green**
W0.0	31:0	**Sample 0 Red**
W1.7:1		Reserved (not written) : W1 is only delivered when **Pixel Fault Mask Enable** is enabled.
W1.0	31:2	Reserved: always written as 0x3fffffff
	1:0	**Pixel Null Mask:** This field has the bit for all samples set to 1 except those pixels in which a null page was source for at least one texel.

Direct Write To Render Target

The 3D Sampler supports the ability to return sample results directly to the Render Target rather than returning them to the requesting agent (e.g. the pixel shader). This feature allows the shader thread which issued the sample message to terminate after sending if the shader is complete. To use this capability, the shader must set the **Response Length** field in the message descriptor to 0 and it must send a message header which sets the **Render Target or Destination Binding Table Index** to the Binding Table Index of the Render Target.

There are a number of important restrictions and considerations:

- As described in the Message Header definition, sample message which require direct write to Render Target require a header. This increases the number of data phases in all message types and may impact performance in some cases.
- Direct write to Render Target by the sampler does not guarantee ordering of writes to the render target from different threads. Multiple threads writing to the same pixel using direct Render Target writes may cause serialization of these threads which may impact performance.

Programming Note	
Context:	Direct Render Target Writes
Sample operations which require direct writes to Render Target must not receive a page-fault response from memory.	

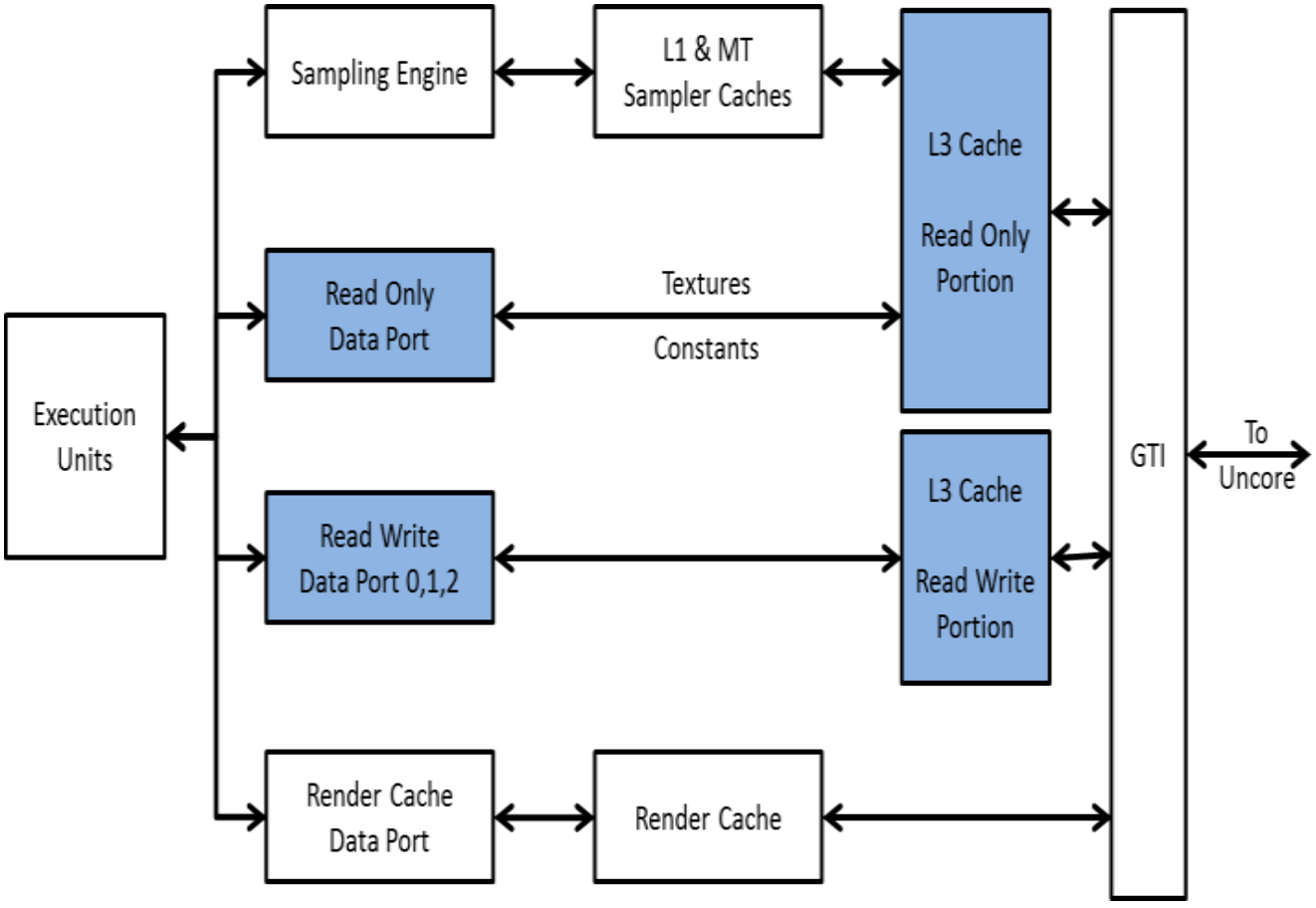
Programming Note	
Context:	SVM
Direct Write to Render Target does not support SVM memory model. Specifically, the Fault and Retry behavior is not supported for direct-write-to-rendert-target shaders.	

Shared Functions – Data Port

The Data Port provides all memory accesses for the Gen 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 Gen 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.

Read, write, and atomic operations issued from the same thread to the same address, using the same data port, the same cache coherency type, and the same page faulting mode, 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, with different cache coherency types, or with different page faulting modes.

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.
- If a virtual address faults, a different memory page access can proceed while the faulted access is handled.

Compared to the Fault-and-Halt page fault mode, the Fault-and-Stream page fault mode is a performance optimization that sometimes enables different order of accesses to continue after a page fault, and still guarantee the order of accesses to the same address within the same thread. Under some circumstances, the hardware automatically treats a Fault-and-Stream page fault as a Fault-and-Halt page fault. See Page Faults in Memory Views for more details on the behaviors of these modes. The Memory Fence message is used to block a thread until all pending Fault-and-Stream page faults are handled when ordering is required.

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 group can use a gateway barrier message with the "Flush to Global Observability" control to guarantee memory ordering at the point when all the threads have reached the barrier. Software can use the results of atomic operations to implement in programs a memory ordering between threads using exclusion locks or other protocols.

Programming Note	
Context:	Read/Write Ordering
In some configurations, a thread group that shares SLM memory can span multiple HDCs. When this occurs, SLM memory ordering is guaranteed by software using a either a memory fence operation or a message barrier	

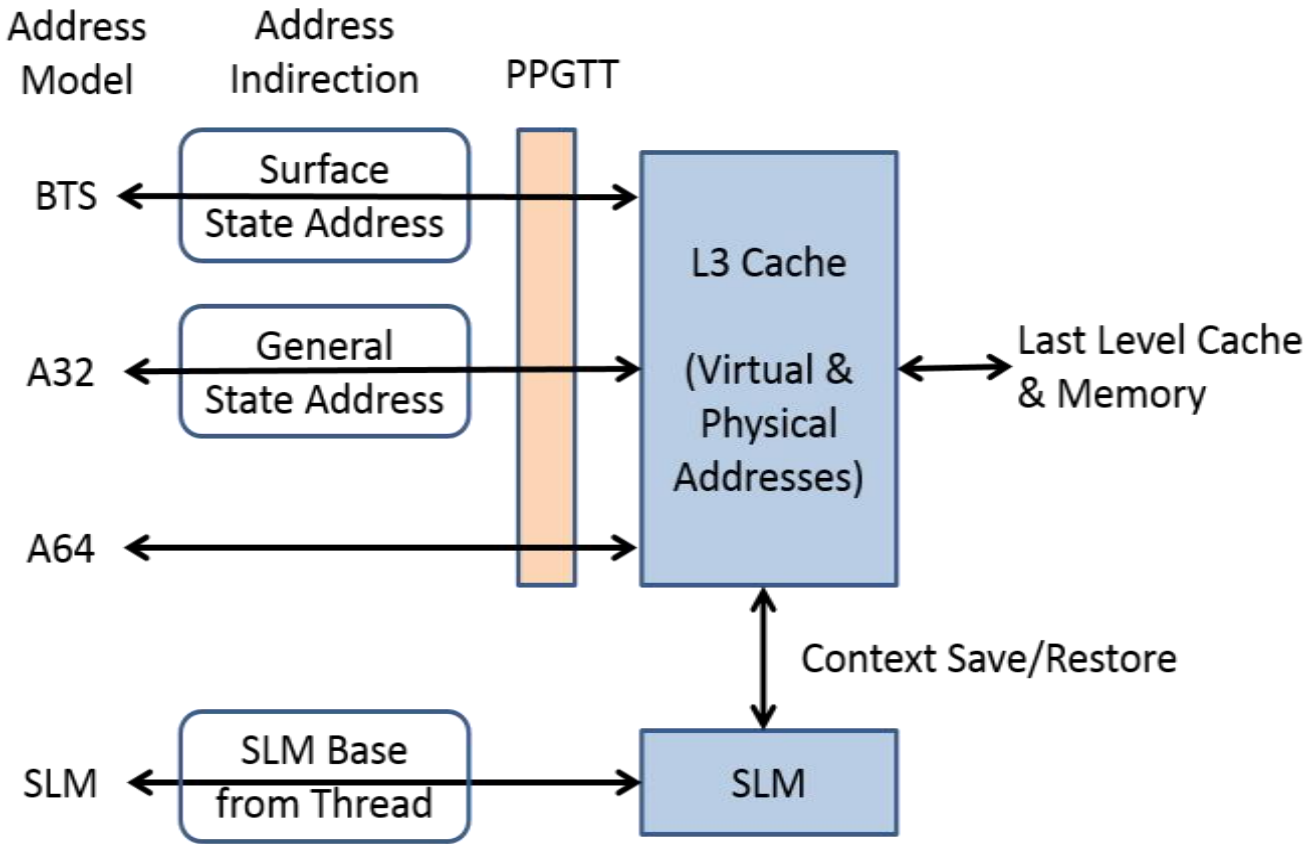
Programming Note	
Context:	Read/Write Ordering
operation to ensure global observability. In configurations where a thread group cannot span multiple HDCs (and when not running in Fault-and-Stream page fault mode), then SLM memory ordering is guaranteed without a fence.	

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. In this model, the Binding Table Index is used to index into the binding table, and the binding table entry contains a pointer to the SURFACE_STATE.

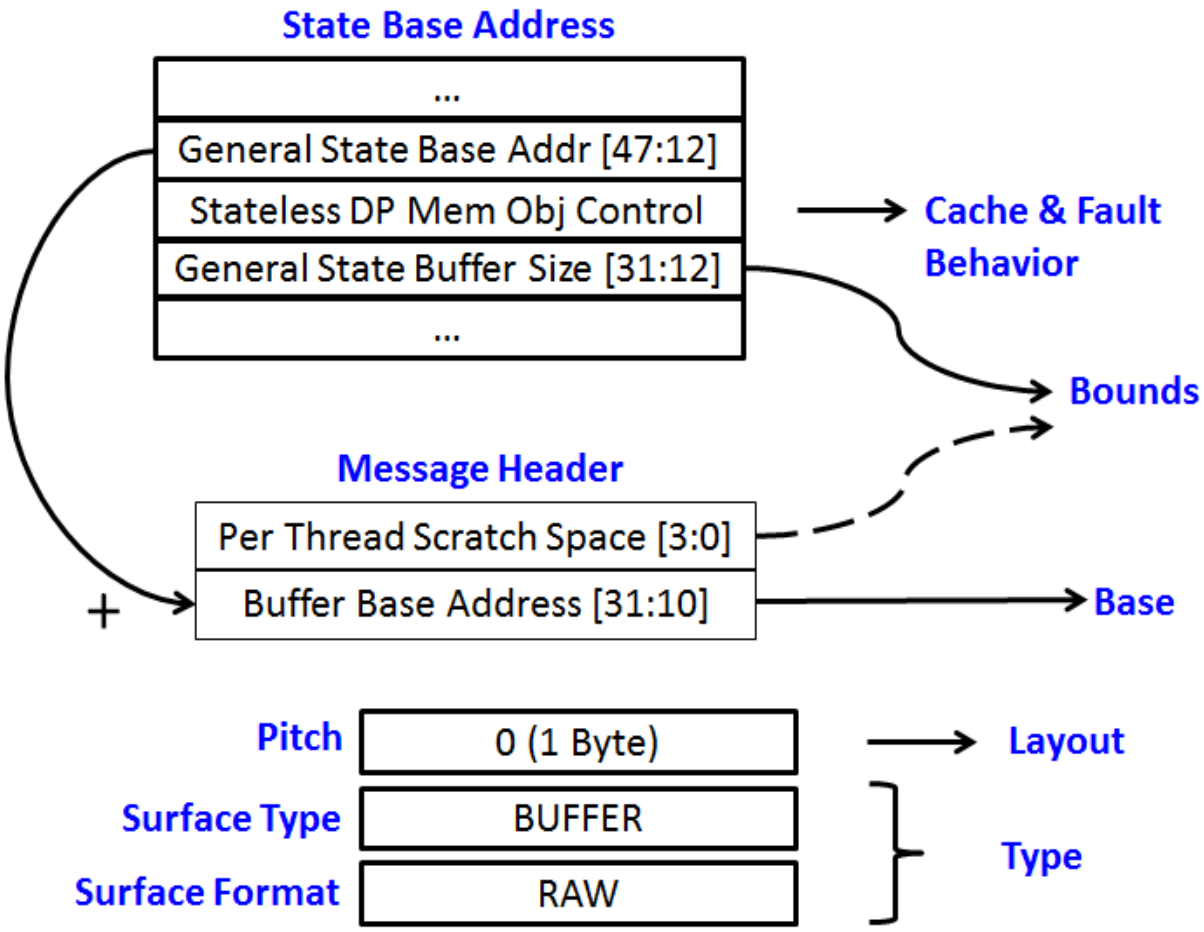
A32 Stateless Model

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 + 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



Stateless Model Aliases

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 Gen and coherent within the entire IA cache memory hierarchy.
253	Non-Coherent	Coherent within Gen for the same Gen cache type (L3, Sampler, or Render).

Programming Note	
Context:	A32 Stateless Model
• 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.	

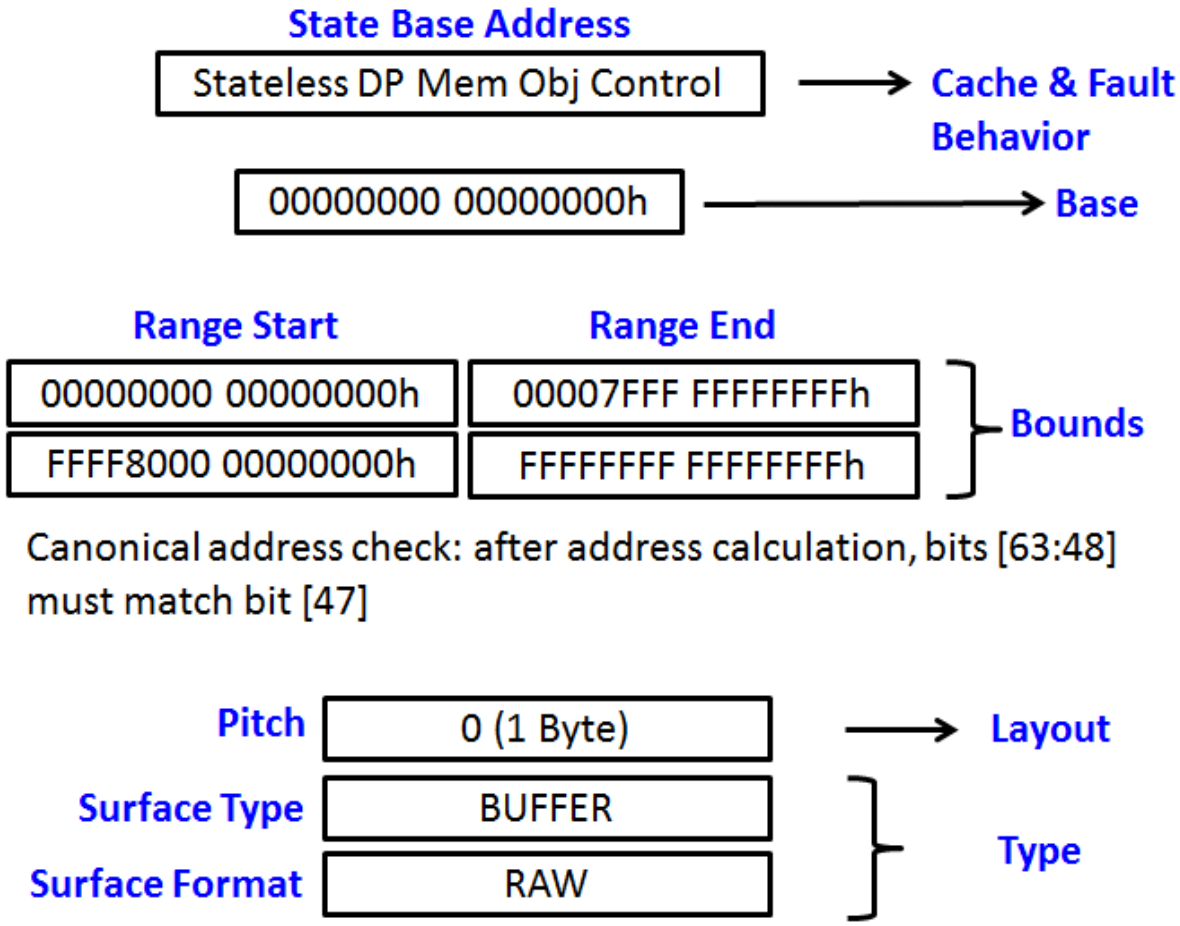
A64 Stateless Model

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



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.	

Shared Local Memory (SLM)

Shared local memory (SLM) is a high bandwidth memory that is not backed up by system memory. It is enabled by configuring the L3 cache to use part of its space for the SLM. 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.

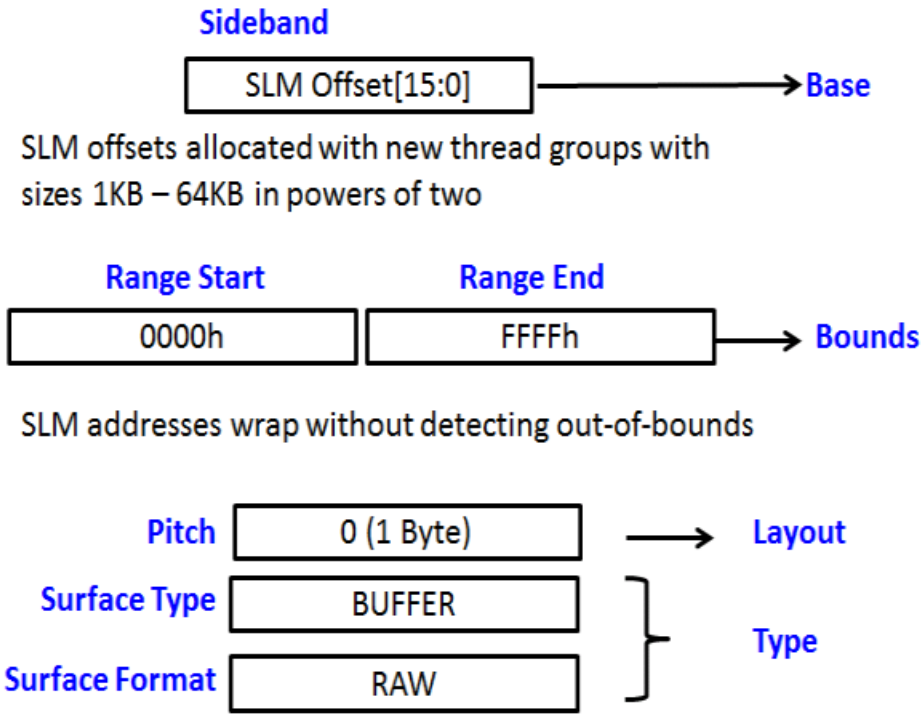
In the SLM model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- 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 is accessed 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.

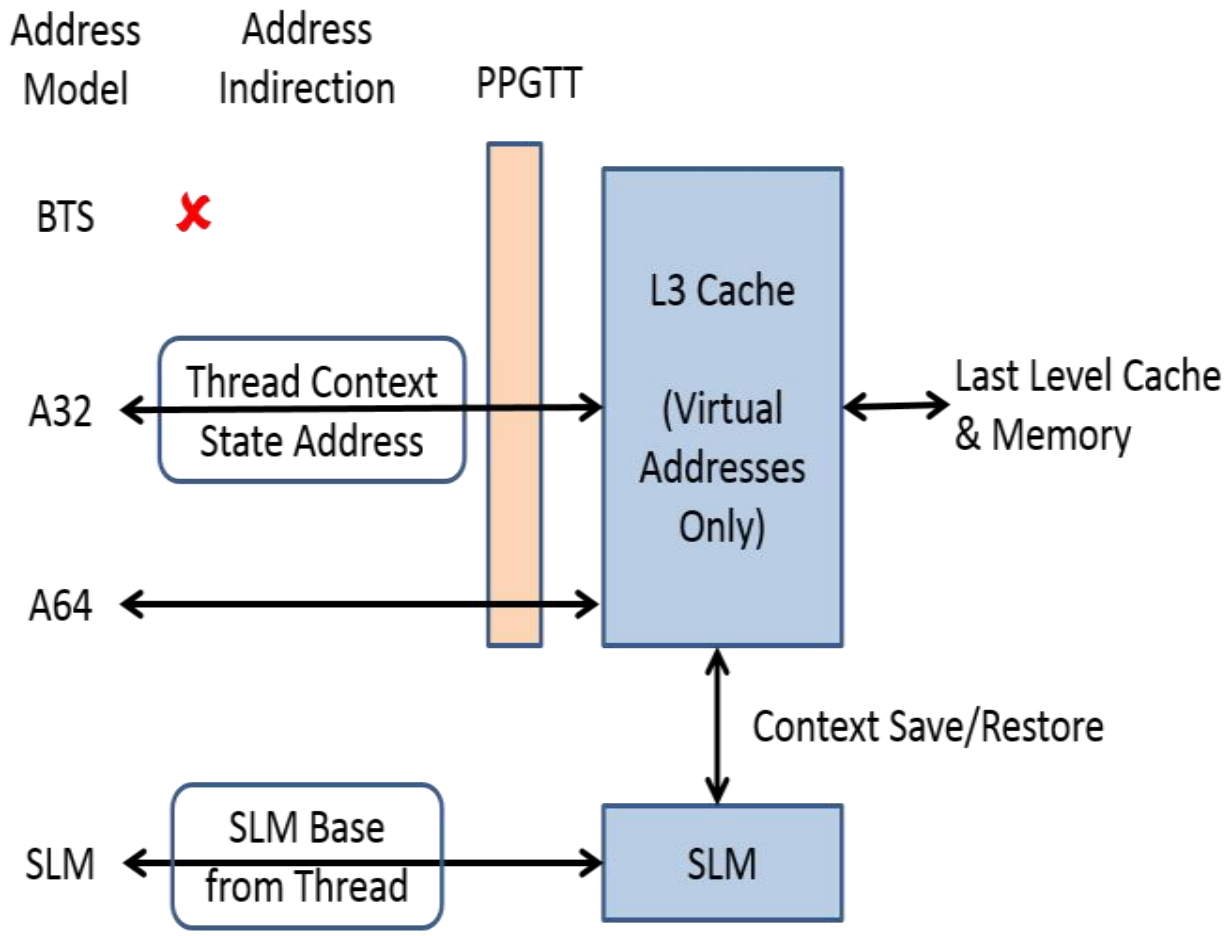
Programming Note	
Context:	Shared Local Memory (SLM)
• Only the data cache data port is supported for SLM, the other data ports treat Binding Table Index 254 as a normal surface state access. • Accesses to SLM do not have any bounds checking. Addresses beyond the size (64 KB) of the SLM wrap around.	

SLM should not be accessed through the 3D pipe.

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 the thread's context save area.

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
Hword Scratch Read/Write messages are the only messages that override A32 base address with the Context Save area's base address during SIP.

Overview of Memory Accesses

The specific message operation modifies the characteristics of the Address Model to determine the behaviors of each Data Port memory access. An overview of the characteristics is provided in the table below, and then explained in more detail in the following sections.

Characteristic	Overview
Surface Type	Each message has its own requirements for which surface types are supported. Most messages only operate on a restricted subset of surface types. Most Data Port messages only operate on the SURFTYPE_BUFFER surface type, but some specific messages operate on other structured surface types.
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 Checking	The calculated virtual memory address, formed from each calculated offset and the Address model's Base Address, could cross the canonical address boundary. Messages using Binding Table State and A32 stateless address models require the graphics driver to correctly set up the base and bounds parameters so that every in-bounds access does not cross the canonical address boundary. Messages using A64 Stateless address models detect any calculated offsets that cross the canonical address boundary, either as a fault or as an out-of-bounds access.
Tiled Resources Checking	The virtual memory address for a data port access may be in the range of addresses that are recognized as Tiled Resources, and that are translated through the TRTT (Tiled Resources Translation Table). Sparse Tiled Resources support a kind of application-managed virtual memory scheme, where the application tells the driver to map specified 64KB address regions within the surface to memory resources called Tile Pools. Tiles that are not mapped to a Tile Pool are either: • Null tiles, which are treated as Out-of-Bounds

Characteristic	Overview
	• Invalid tiles, which are treated as page faults Tiles that are mapped to a Tile Pool have a virtual address that is handled like any virtual address that is outside of the Tiled Resources address range.
Fault Handling	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. SLM accesses do not fault. See Read/Write Ordering in this volume for a description of how memory ordering is impacted by a page fault.
Out-of-Bounds handling	If a memory access is marked as out of bounds for the surface or the message, or if a virtual address maps to a Tiled Resource Null page, then a read operation returns 0 and a write operation is dropped.
Cache Behavior	Data port memory values can be cached both inside or outside the GPU. The primary GPU data cache can either be kept coherent with the IA processor with every memory acess, or left non-coherent where software protocols manage the coherency. Each Address Model specifies the coherency model for the read/write data cache ports. • For Surface State accesses: the surface's RENDER_SURFACE_STATE **Coherency Type** field • For A32 and A64 Stateless accesses: the 253 and 255 aliases (see Stateless Model Aliases in A32 Stateless Model in this volume for details) • SLM accesses are localized and not cached The behavior of writing cache data to memory (e.g. write back or write through modes) is specified by the MEMORY_OBJECT_CONTROL_STATE. • For Surface State accesses: the surface's RENDER_SURFACE_STATE **Surface Object Control State** • For A32 and A64 Stateless accesses: the STATE_BASE_ADDRESS **Stateless Data Port Access Memory Object Control** When PPGTT is enabled, the page table entry overrides the LLC/eDRAM cache controls supplied by the MEMORY_OBJECT_CONTROL_STATE.

Surfaces Types

Each message has its own requirements for which surface types are supported. Most messages only operate on a limited subset of surface types. The BUFFER surface type is supported by all the address models. All other surface types are only supported by the BTS address model.

The stateless and SLM address models have a pre-defined surface type, format, and layout. Untyped messages are used with these address models.

The BTS address model is used with either typed or untyped messages. The BTS surface type must be compatible with the expected surface type, format, and layout of the message operation. Untyped messages override the surface format with their specific data type and size. Typed messages require the surface characteristics expected by the message, or the message results are undefined (accesses may be dropped or may return random data).

Surface Types used by Data Port Messages

Message Category	BTS	SLM	A32	A64
Render Target	1D, 2D, 3D, CUBE, 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, STRBUF, 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 STRBUF 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			
STRBUF	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

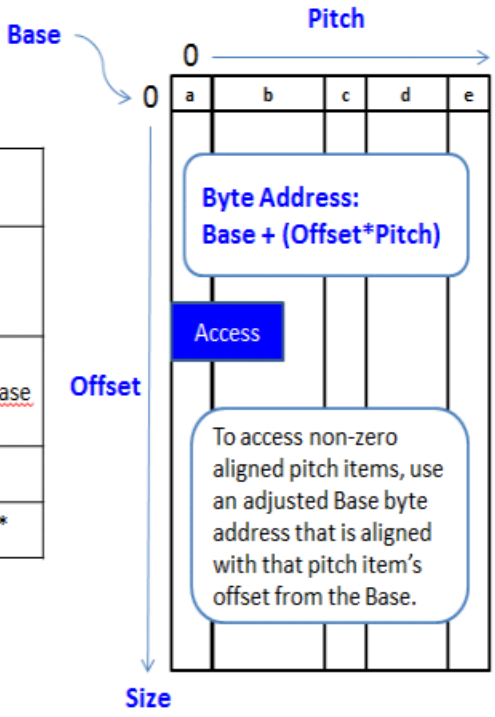
Bounds Check:
- **Offset < (Buffer Size – ((Access Width - 1) / Pitch))**

Address Model	BTS	A32	A64	SLM
Base Address Alignment	32B	2KB	0	1KB
Buffer Size	Surface Size*	General State Buffer Size	Canonical = 47 bits	64KB – SLM_Base
Offset Size	32 bits	32 bits	64 bits	32 bits
Pitch Size	11 bits	16 bits**	16 bits**	7 bits**

* Surface Size - 1 =

Depth	Height	Width
U11	U14	U7

** Pitch Size specified in Data Port 2 scaled messages



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).

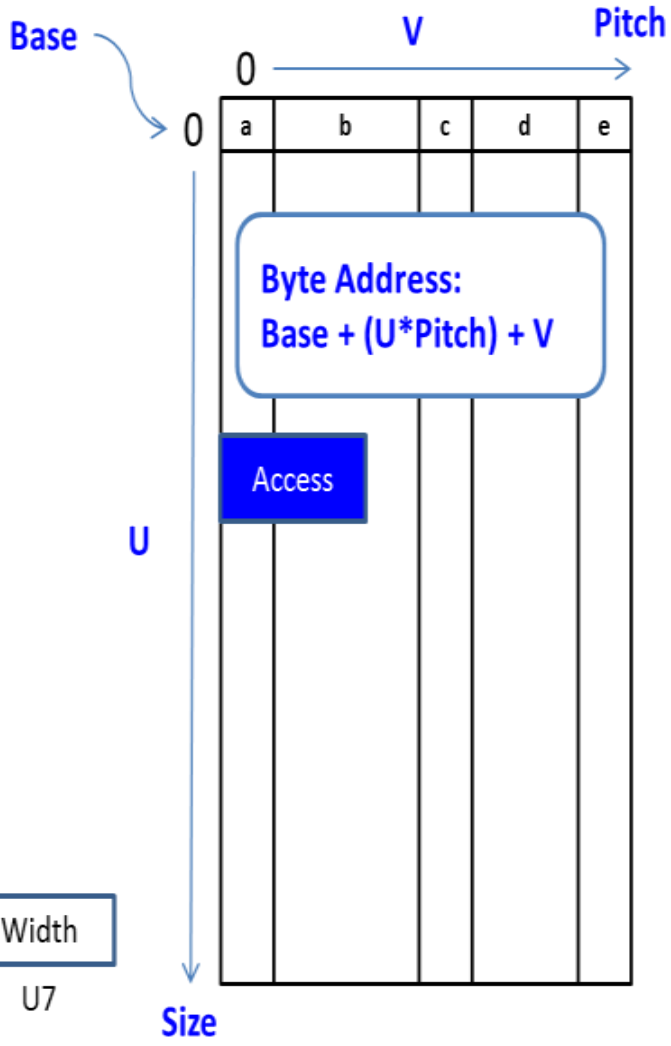
Addressing Structured Buffers (SURFTYPE_STRBUF)

Some data port messages operate on STRBUF surface types. The figure below illustrates how the base address and bounds calculations are performed on BUFFER surface types.

STRBUF: Array of Structures

Bounds Check:
- **U < Size**
- **V < Pitch – Access Width**

Address Model	BTS
Base Address Alignment	32B
Buffer Size	Surface Size*
U & V Size	32 bits
Pitch Size	11 bits



* Surface Size - 1 =

Depth	Height	Width
U11	U14	U7

Programming Note	
Context:	Addressing Structured Buffers (SURFTYPE_STRBUF)
Read/Write Data Ports only support linear tiled STRBUF.	

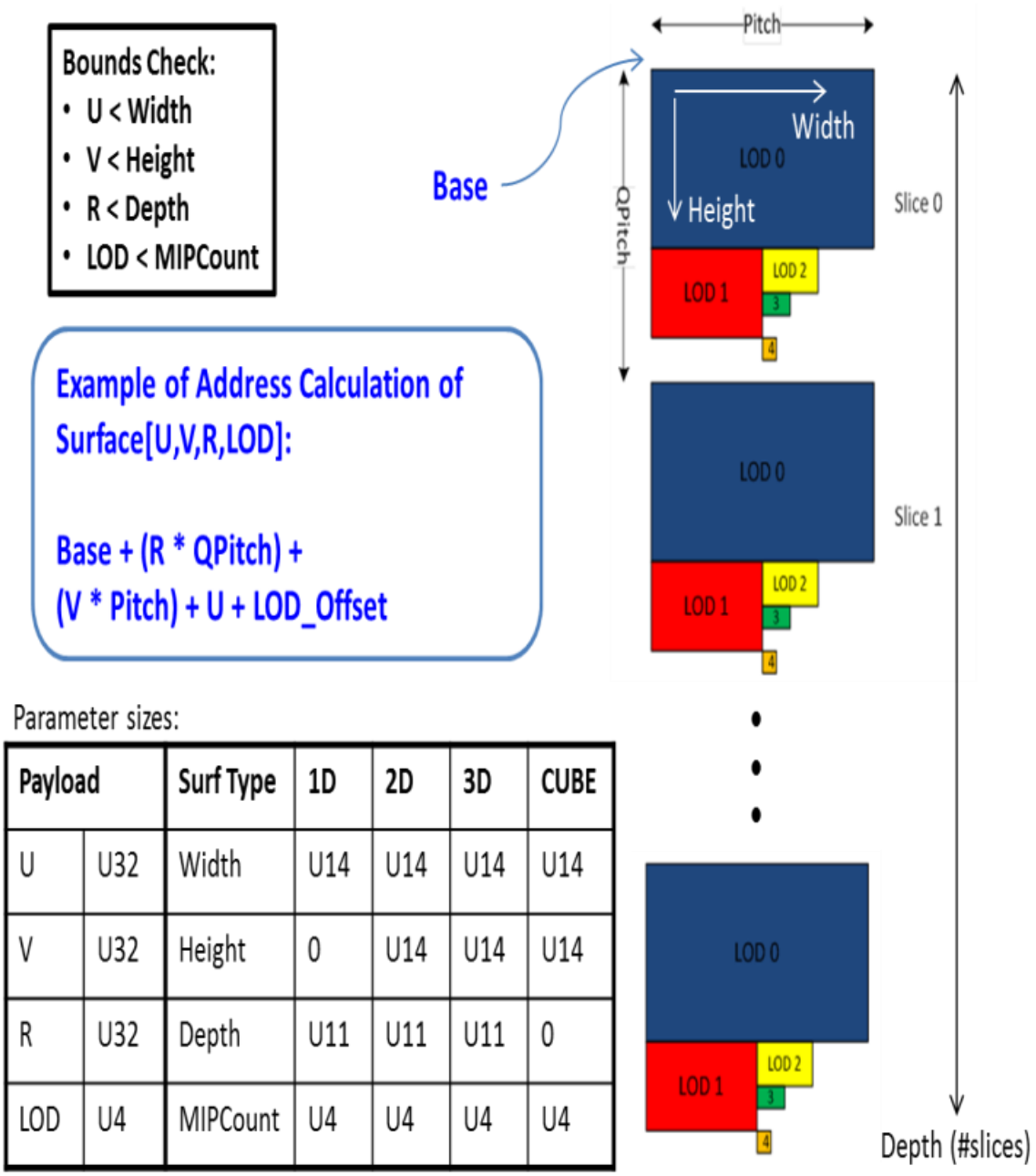
For untyped messages accessing SURFTYPE_STRBUF surface types, both the V address and Surface Pitch must be aligned to the data access width (for example, DWords must have the low 2 bits as zeros).

Addressing 1D, 2D, 3D, CUBE Surfaces

Some data port messages operate on various BTS surface types. The addressing and bounds checking for the 1D, 2D, 3D, and CUBE surface types is fully explained in other chapters. See SURFACE_STATE in the Shared Functions volume and Surface Layout and Tiling in the GPU Overview volume.

The figure below illustrates the differences in addressing and bound checking between these surface types and the BUFFER and STRBUF surface types.

Multi-Dimensional Surface Buffers



Parameter sizes:

Payload		Surf Type	1D	2D	3D	CUBE
U	U32	Width	U14	U14	U14	U14
V	U32	Height	0	U14	U14	U14
R	U32	Depth	U11	U11	U11	0
LOD	U4	MIPCount	U4	U4	U4	U4

The following table indicates the hardware interpretation of each input address parameter based on surface type. If LOD is specified, then both V and R parameters must also be included, even if they are ignored. If LOD is not specified, then unused V and R parameters are optional in the address payload. If LOD is not present in the address payload, then it has the default value of 0. If the surface state indicates the Number of Multisamples > 1, then the sample number is the first parameter in the address payload.

Address Parameters Used by Surface Type

SURFACE_STATE Fields			Address Payload				
Surface Type	Surface Array	Number of Multisamples	MSAA	U Address	V Address	R Address	LOD
SURFTYPE_1D	disabled	Must be MULTISAMPLECOUNT_1	Never Present	X pixel address	(ignored if present)	(ignored if present)	Optional LOD
	enabled			X pixel address	array index	(ignored if present)	Optional LOD
SURFTYPE_2D	disabled	MULTISAMPLECOUNT_1	Never Present	X pixel address	Y pixel address	(ignored if present)	Optional LOD
	enabled			X pixel address	Y pixel address	array index	Optional LOD
	disabled	Not MULTISAMPLECOUNT_1	Sample Number	X pixel address	Y pixel address	(ignored if present)	Optional LOD
	enabled			X pixel address	Y pixel address	array index	Optional LOD
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		

Workaround	
Context:	Addressing 1D, 2D, 3D, CUBE Surfaces

If the surface state indicates the Number of Multisamples > 1, then the LOD parameter is not optional: the R and LOD parameters must be specified along with the MSAA sample number parameter.

Programming Note	
Context:	Addressing 1D, 2D, 3D, CUBE Surfaces

- Vertical stride & Vertical Offset fields of the surface state object are only supported for 2D non-array surfaces.
- Tile W surfaces must be of format R8_UINT and only support SIMD8.

Surface Formats

Untyped messages allow direct read and write access to untyped surfaces. Untyped surfaces are always type BUFFER or STRBUF, and are assumed to be surface format RAW.

Typed messages allow direct read and write accesses to typed surfaces. Typed surfaces are of type SURFTYPE_1D, 2D, 3D, CUBE, or BUFFER and have a supported surface format other than RAW.

Read and write messages for typed surfaces convert the message's data type to/from the selected surface's format. The conversion rules are listed in a table below.Some surface formats are supported for typed surface writes, but are indirectly supported for typed surface reads. Indirectly supported surface formats provide the RAW pixel data in UINT formats, which enables software to complete the conversion to their defined format.

The read conversion of typed surface format to a register data format is always exact. If the destination precision is less than the source precision, a write conversion from a register data format to a typed surface format may be rounded, and denormalized or clamped.

Conversion Rules for Typed Surface Dataport Messages

Register Data Type	Surface Format Type	Read Conversion (Surface to Register)	Write Conversion (Register to Surface)
F32	FLOAT	IEEE float conversion. Normalize denorms if source is narrower. Convert SNan to QNan quietly.	IEEE float conversion. Round to even and denormalize if destination is narrower. Convert SNan to QNan quietly.
F32	SNORM, UNORM	Convert fixed point to IEEE float	Convert IEEE float to fixed point. Round to even and clamp to min/max if destination is narrower.
S31	SINT	Sign extend MSBs if source is narrower	Clamp to min/max if destination is narrower.
U32	UINT	Zero extend MSBs if source is narrower	Clamp to min/max if destination is narrower.
RAW32	Any	32 bit copy	*Not supported*
RAW16	Any	16 bit copy, zero extend MSBs	*Not supported*
RAW8	Any	8 bit copy, zero extend MSBs	*Not supported*

Typed dataport messages support a subset of all surface state formats. The formats supported by typed surface reads and writes are listed in the table below. The table shows what values and formats are returned for the components when read.

Typed surface formats with UINT require the message data in U32 format. Surface formats with SINT require the message data in S32 format. All other typed surface formats require the message data in FLOAT32 format. The surface format for the typed atomic integer operations must be R32_UINT or R32_SINT.

All typed surface formats are supported for writes, as long as the Bits Per Element is a multiple of 8 (byte). This includes all the surface formats excluding SRGB and YCRV formats. The table shows the full list of supported surface formats for writes.

For typed surface writes where the Surface Format has components that are not byte-aligned, each shader channel select in the surface state must be set to a unique surface channel (SCS_RED, SCS_GREEN, SCS_BLUE, SCS_ALPHA) and the value of (SCS_ZERO, SCS_ONE) cannot be selected. Also 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 Write and 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	RAW32	RAW32	0	1
081	R16G16B16A16_SNORM	64	RAW32	RAW32	0	1
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
0C4	R10G10B10A2_UINT	32	*RAW32*	0	0	1
0C7	R8G8B8A8_UNORM	32	RAW32	0	0	1
0C9	R8G8B8A8_SNORM	32	RAW32	0	0	1
0CA	R8G8B8A8_SINT	32	S31	S31	S31	S31
0CB	R8G8B8A8_UINT	32	U32	U32	U32	U32
0CC	R16G16_UNORM	32	RAW32	0	0	1
0CD	R16G16_SNORM	32	RAW32	0	0	1
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

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)	Data Returned by Typed Read			
			R	G	B	A
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	RAW16	0	0	1
107	R8G8_SNORM	16	RAW16	0	0	1
108	R8G8_SINT	16	S31	S31	0	1
109	R8G8_UINT	16	U32	U32	0	1
10A	R16_UNORM	16	RAW16	0	0	1
10B	R16_SNORM	16	RAW16	0	0	1
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	RAW8	0	0	1
141	R8_SNORM	8	RAW8	0	0	1
142	R8_SINT	8	S31	0	0	1
143	R8_UINT	8	U32	0	0	1
144	A8_UNORM	8	RAW8	0	0	1

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 read and write messages, and the Shader Channel Select fields from the surface state, are interpreted with these same substitute formats.	

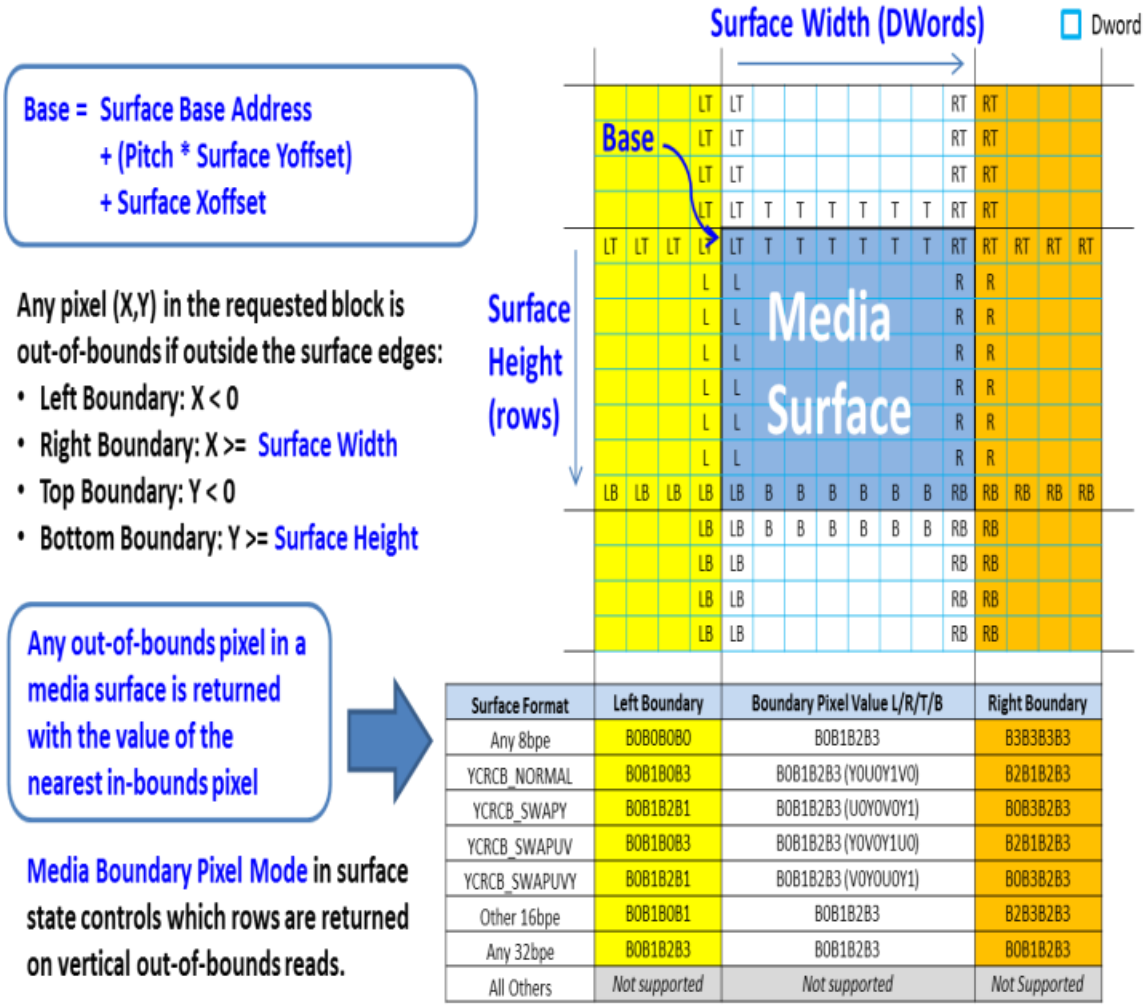
Addressing 2D Media Surfaces

Some data port messages operate on rectangular blocks of 2D surfaces using the BTS address model. These messages are typically used in media image processing. The rectangular block is specified in media messages as a signed (X, Y) offset from the surface's origin, and by the block's width and height.

The rectangular block specified in a media message could be partially (or fully) out-of-bounds of the surface. Some media data port messages perform pixel replication to supply values for out-of-bounds

pixels. The figure below illustrates how the boundary pixel values are calculated from an in-bounds pixel value.

Media Surface Boundary Pixel Handling



Surface Format	Left Boundary	Boundary Pixel Value L/R/T/B	Right Boundary
Any 8bpe	B0B0B0B0	B0B1B2B3	B3B3B3B3
YCRCB_NORMAL	B0B1B0B3	B0B1B2B3 (Y0U0Y1V0)	B2B1B2B3
YCRCB_SWAPY	B0B1B2B1	B0B1B2B3 (U0Y0V0Y1)	B0B3B2B3
YCRCB_SWAPUV	B0B1B0B3	B0B1B2B3 (Y0V0Y1U0)	B2B1B2B3
YCRCB_SWAPUVY	B0B1B2B1	B0B1B2B3 (V0Y0U0Y1)	B0B3B2B3
Other 16bpe	B0B1B0B1	B0B1B2B3	B2B3B2B3
Any 32bpe	B0B1B2B3	B0B1B2B3	B0B1B2B3
All Others	*Not supported*	*Not supported*	*Not Supported*

Media accesses are Dwords: all 4 surface boundaries are Dword aligned
(Base is 32B aligned, Pitch is 64B aligned, Xoffset is 4*bpe aligned, Width is Dword aligned).

Upper left corner of the requested Block is specified by the signed (X, Y) coordinate.
The block offset and block width are always aligned to size of pixel (BPE) in surface.

The addressing of 2D Media surfaces is subset of the generalized surface state addressing model. For more information, see SURFACE_STATE in the Shared Functions volume and Surface Layout and Tiling in the GPU Overview volume.

The Vertical Stride and Vertical Stride Offset fields of the surface state object are used for 2D media surfaces to support interlaced formats in a full frame.

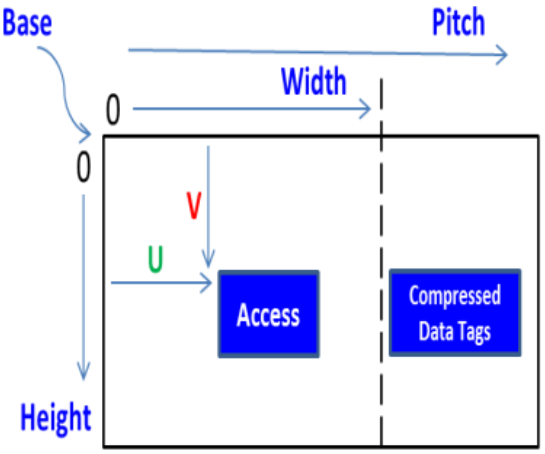
The surface's tile mode specifies how the 2D surface is laid out in the linear virtual address space. The figure below illustrates how pixel's byte address is calculated in a 2D media surface. The table lists how the surface's linear virtual address offset A[38:0] is calculated.

Media Surface Addressing

Guarantee the access is in surface bounds
with Media Boundary Pixel Handling:

- Clamp X in [0, Width-1]
- Clamp Y in [0, Height-1]
- U = Surface Xoffset + (4 * Clamp X)
- V = Surface Yoffset + Vertical Line Stride Offset
  + (Vertical Line Stride * Clamp Y)

Byte Address: Surface_Base + Tiled_2D_Surface_Address_Offset



Media surfaces are optionally compressed, with compressed data tags in pitch region

Bits per Element	Tile Mode	Tile constants		Tiled 2D Surface Address Offset																	
		Cv	Cu	A[38:16]	A[15:12] (64KB Tile)				A[11:6] (4KB Tile)						A[5:0] (Cache Line)					
64 & 128	TileYS	6	10	u[13:10] + (v[13:6]*Pitch[17:10])	u9	v5	u8	v4	u7	v3	u6	v2	u5	u4	v1	v0	u3	u2	u1	u0
	TileYF	4	8	u[13:8] + (v[13:4]*Pitch[17:8])																
16 & 32	TileYS	7	9	u[13:9] + (v[13:7]*Pitch[17:9])	u8	v6	u7	v5	u6	v4	u5	v3	u4	v2						
	TileYF	5	7	u[13:7] + (v[13:5]*Pitch[17:7])																
8	TileYS	8	8	u[13:8] + (v[13:8]*Pitch[17:8])	u7	v7	u6	v6	u5	v5	u4	v4	u3	v2						
	TileYF	6	6	u[13:6] + (v[13:6]*Pitch[17:6])																
all	TileY	5	7	u[13:7] + (v[13:5]*Pitch[17:7])					u6	u5	u4	v4	u3	v2						
all	TileX	3	9	u[13:9] + (v[13:3]*Pitch[17:9])					v2	v1	v0	u8	u7	u6	u5	u4	u3	u2	u1	u0
all	Linear	0	0	u[13:0] + (v[13:0]*Pitch[17:0])																

When a 2D surface is tiled, the Surface Base Address is aligned on the tile size (4KB or 64KB).
NV12 surfaces use X/Y Offset for UV Plane surface state so that Luma 8bpe and Chroma 16bpe both aligned for TileYF/YS.

2D Media surfaces can be set up to be compressible using the Memory Compression Enable surface state field. Some media hardware will write blocks in a compressed format. The media dataport messages will read those compressed formats and automatically decompress a compressed location when read. The tag information needed to interpret and decompress data is stored in the memory region between the surface width and pitch.

Programming Note	
Context:	Addressing 2D Media Surfaces
If the surface state field Memory Compression Enable is set, then the surface state field Coherency Type must be set to GPU Coherent (not IA coherent).	

Programming Note	
Context:	Addressing 2D Media Surfaces
Writing a compressible media surface with any dataport message is not supported and produces undefined results. Reading a compressible media surface using any dataport message other than Media Block Read is not supported and produces undefined results.	

Programming Note	
Context:	Addressing 2D Media Surfaces

- The surface base address must be 32-byte aligned.
- The surface width must a multiple of DWords.
- Pitch must be a multiple of 64 bytes when the surface is Linear.
- Media block writes to Linear or TileX surfaces must have a height of 16 or less.
- For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. DWord-aligned).
- For media block writes, both X Offset and Block Width must be DWord-aligned.
- The block width and offset must be aligned to the size of pixels stored in the surface. For a surface with 8bpe pixels for example, the block width and offset can be byte-aligned. For a surface with 16bpe pixels, it is word-aligned.

2D Media Surface Formats

Media dataport messages allow direct read and write accesses to media surfaces. Media surfaces are only type SURFTYPE_2D.

Read and write messages for media surfaces never convert the data type of the selected surface's format.

If the media surface state is set up as not compressible, the formats supported are listed in the table below. Reads or writes to an unsupported surface format have undefined results. The surface format is only used to determine out-of-bounds pixel replication.

If the media surface state is set up as compressible, then only a small subset of surface state formats are supported by the media data port message. The formats supported by compressible media surface reads are listed in the table below. Reads to an unsupported surface format return undefined results. Writes to a compressible surface format have undefined results. The surface format is used to determine out-of-bounds pixel replication and to determine compression type.

Supported Media Surface Formats

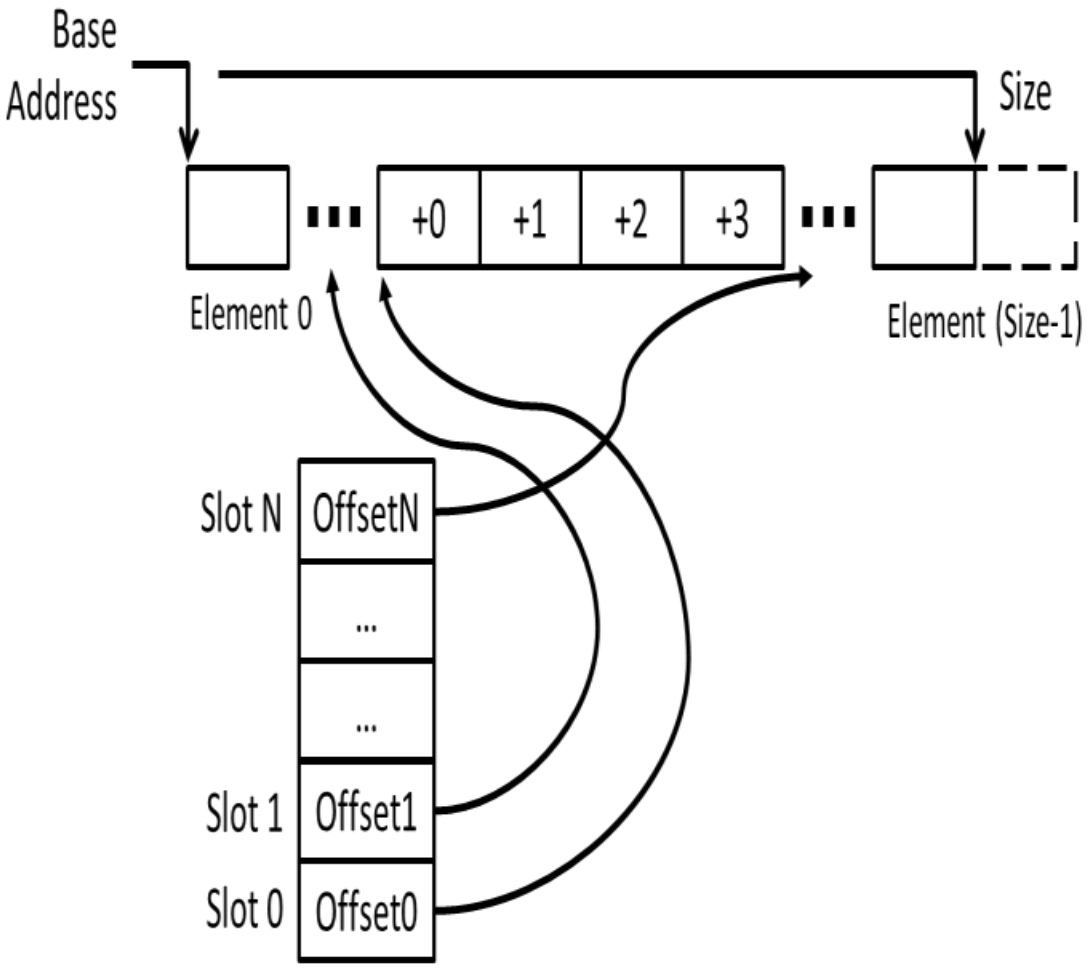
Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
0C0	B8G8R8A8_UNORM	32	*Not supported*
0C2	R10G10B10A2_UNORM	32	*Not supported*
0C4	R10G10B10A2_UINT	32	*Not supported*
0C7	R8G8B8A8_UNORM	32	*Not supported*
0C9	R8G8B8A8_SNORM	32	*Not supported*
0CA	R8G8B8A8_SINT	32	*Not supported*
0CB	R8G8B8A8_UINT	32	*Not supported*
0CC	R16G16_UNORM	32	*Not supported*
0CD	R16G16_SNORM	32	*Not supported*
0CE	R16G16_SINT	32	*Not supported*
0CF	R16G16_UINT	32	*Not supported*
0D0	R16G16_FLOAT	32	*Not supported*
0D1	B10G10R10A2_UNORM	32	*Not supported*
0D3	R11G11B10_FLOAT	32	*Not supported*
0D6	R32_SINT	32	*Not supported*
0D7	R32_UINT	32	*Not supported*
0D8	R32_FLOAT	32	*Not supported*
100	B5G6R5_UNORM	16	*Not supported*
102	B5G5R5A1_UNORM	16	*Not supported*
104	B4G4R4A4_UNORM	16	*Not supported*
106	R8G8_UNORM	16	NV12 (UV 8bpe)
107	R8G8_SNORM	16	*Not supported*
108	R8G8_SINT	16	*Not supported*
109	R8G8_UINT	16	NV12 (UV 8bpe)
10A	R16_UNORM	16	Y16
10B	R16_SNORM	16	*Not supported*
10C	R16_SINT	16	*Not supported*
10D	R16_UINT	16	Y16
10E	R16_FLOAT	16	*Not supported*
113	A16_UNORM	16	Y16
11A	B5G5R5X1_UNORM	16	*Not supported*
140	R8_UNORM	8	NV12 (Y), Y8, IMC1 (YUV), IMC3 (YUV)
141	R8_SNORM	8	*Not supported*
142	R8_SINT	8	*Not supported*

Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
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).

Data port messages have different semantics on how the execution mask controls the read or write accesses. The table below summarizes the semantics for all the data ports. The details of which semantics are used by each individual message is described in the Messages section of this volume.

Description of SIMD Execution Mask Semantics

Execution Mask Name	SIMD Mode and Data Type	How Execution Mask is Used
SM	SIMD8 or SIMD16	For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels. For SIMD8, the lower 8 bits of the execution mask control the 8 SIMD channels, and the upper 8 bits of the execution mask are ignored.
SMBLK	SIMD8 or SIMD16 With Data Elements	For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels. For SIMD8, the lower 8 bits of the execution mask control the 8 SIMD channels, and the upper 8 bits of the execution mask are ignored. If the date elements > 1, then the execution mask bits are reused for the additional data elements.
SMPSM	SIMD8 or SIMD16 Pixel Surface	For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels. For SIMD8, the lower 8 bits of the execution mask control the 8 SIMD channels, and the upper 8 bits of the execution mask are ignored. The bits of the execution mask are ANDed with the corresponding bits of the Pixel/Sample Mask from the message header, and the resulting mask determines which slots are enabled. If the message header is not present, only the execution mask is used.

Execution Mask Name	SIMD Mode and Data Type	How Execution Mask is Used
SM4x2	SIMD4x2	The lower 8 bits of the execution mask control the 2 SIMD channels. The lower 4 bits control the first channel and the upper 4 bits control the upper channel. The upper 8 bits are ignored. For reads and writes, if any one or more bits within the corresponding execution mask nibble are enabled, the channel is enabled.
SG	SIMD8 Slot Group	Either the lower 8 bits or the upper 8 bits of the execution mask are used to control the 8 SIMD channels, based on the slot group specified in the message descriptor.
SGPSM	SIMD8 Slot Group on Pixel Surface	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. The bits of the execution mask are ANDed with the corresponding bits of the Pixel/Sample Mask from the message header, and the resulting mask determines which slots are enabled. If the message header is not present, only the execution mask is used.
BLK	Block	The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords. For reads, the enables operate in OWord channel mode: if any one or more asserted bits within the OWord's corresponding execution mask nibble are enabled, the whole OWord is read regardless of the setting of any particular DWord's bit. For writes, the enables operate in DWord channel model: the DWords are written if the corresponding DWord's execution mask bit is set. If the block is 1 OWord size, then either the lower or upper nibbles of the lower 8 bits control the corresponding DWords in that lower or upper OWord. If the block is 8 OWords or 16 OWords, then the execution mask bits are reused for the additional channels.
DBLK	Dual Block	The lower 8 bits of the execution mask control the 8 SIMD DWord channels. The lower 4 bits control the first channel and the upper 4 bits control the second channel. The upper 8 bits are ignored. For reads, the enables operate in OWord channel mode: if any one or more asserted bits within the OWord's corresponding execution mask nibble are enabled, the whole OWord is read regardless of the setting of any particular DWord's bit. For writes, the enables operate in DWord channel model: the DWords are written if the corresponding DWord's execution mask bit is set. If the block is 8 OWords (4 dual blocks), then the lower 8 execution mask bits are

Execution Mask Name	SIMD Mode and Data Type	How Execution Mask is Used
		reused for the additional OWord blocks 1-3.
BLKCM	Block with Channel Mode	The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords. For DWord channel mode, the execution mask delivered with the message dictates DWord-based control of read or write operations. For OWord channel mode, any one or more asserted bits within the OWord's corresponding execution mask nibble causes read or write operations to occur across all four DWords of the OWord regardless of the setting of any particular DWord's bit. If the block is 1 OWord size, then either the lower or upper nibbles of the lower 8 bits control the corresponding DWords in that lower or upper OWord. If the block is 8 OWords or 16 OWords, then the execution mask bits are reused for the additional blocks.
Ignored	Not Used	The execution mask bits are ignored by this message operation. The data is controlled by other parameters to this message. The execution mask bits must be non-zero for the message to be sent to the data port.

Address Alignment and Data Widths

Data Port messages calculate multiple data operand addresses using offsets from the message's common base address. The details of how each message calculates its address is described in the Messages section.

The address calculations are described using the following notational convention:

(Base + BaseOffset) → DataType[Offset]

This notation refers to the byte-aligned address (**Base + BaseOffset**), which is used as the address of a linear array of **DataType** operands. The **Offset** is the index into the **DataType** array with that data width.

Address Calculation

Notation	Description
Base	Each Address Model has its own base address source, illustrated in the Surfaces Types section of this volume. The Base address is architecturally defined to always be aligned at a valid address boundary for the message. The notational convention in this chapter labels this calculated base address the **Base**.

Notation	Description
Buffer, BaseOffset	The offsets from some messages are specified as byte offsets, and others are DWord (or other DataType) offsets. **Buffer** is shorthand for Buffer Base Address offset that is specified in A32 message headers. The notation in this chapter shows byte offsets as being added to the **Base** on the left side of the pointer (→), and the DataType-sized offsets as being indices to the array on the right of the pointer. Each message defines the address alignment it operates on. Base address offsets are forced to be aligned for the message's DataType by treating any unaligned low address bits as zero. For most messages (except media block messages), the Buffer or BaseOffset value is always an unsigned (positive) value.
B[ ], W[ ], DW[ ], QW[ ], OW[ ], HW[ ]	The **DataType** array is a linear array of Bytes (**B**), Words (**W**), DWords (**DW**), QWords (**QW**), OWords (**OW**), or HWords (**HW**). The index of the array is multiplied by the width of the data type, to form a byte offset to be added to the base address. Each message defines the data type it operates on. The array's byte offset is architecturally defined to always be aligned at a valid address boundary for the message.
Offset{Slot}, U{Slot}, V{Slot}	**Offset{Slot}** is shorthand for Offset0, Offset1, … in a SIMD8 or SIMD16 address payload. **U{Slot}** and **V{Slot}** are shorthand for saying U.Offset{Slot} and V.Offset{Slot}, where U and V are SIMD8 or SIMD16 address payloads. For most messages (except media block messages), the **Offset** is always an unsigned (positive) value.
Elem, Chan	**Elem** represents the index into an array of consecutive data operands. **Chan** represents the 32-bit Red, Green, Blue, or Alpha channels of color item. These values are sequentially stored in memory as a 4 item DW block.
(Surface[U, V, R, LOD])	The notation **Surface[U, V, R, LOD]** refers to a typed surface which, depending on the surface type, may use the U, V, R, and LOD parameters to calculate the offsets from the surface's state base address for the operands.

Bounds Checking and Faulting

Bounds checking is a programming safety feature of the Gen 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.
Surface	BTS	STRBUF	Messages using the BTS address model detect any U address component that exceeds the U size of the structured buffer, or a V address component that exceeds the Pitch of the structured buffer, as specified in the SURFACE_STATE.
Surface	BTS	1D, 2D, 3D, CUBE	Messages using the BTS address model detect any U, V, or R address components that exceed the Width, Height, and Depth sizes of the surface, as specified in the SURFACE_STATE.
Media	BTS	2D	Some media-oriented messages using the BTS address model detect X or Y address components that exceed the Width and Height of the surface, as specified in the SURFACE_STATE. If a read access is outside of the surface boundaries, the read access is treated as an in-bounds access and returns the value of the nearest pixel. Write accesses outside of the surface boundaries are dropped.
GenState	A32	BUFFER	Messages using the A32 Stateless address model detect any address offsets that, either directly or through the offset calculations, exceed the General State Buffer Size specified in the STATE_BASE_ADDRESS. If General State Buffer Size is zero, then any A32 Stateless access is out-of-bounds.
PTSS	A32	BUFFER	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,

Bounds Check	Address Model	Surface Type	Description
			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 calculated address offsets that cross the canonical address boundary as out-of-bounds. Any A64 offsets that are used directly as the byte address without scaling do not require a bounds check. 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.
Wrap	SLM	BUFFER	Messages using the SLM address model do not detect any address offsets that, either directly or through the offset calculations, exceed the SLM memory size. If the SLM memory size is exceeded, the SLM address wraps around in the SLM memory and is treated as an in-bounds access.

Message Formats

Message operations on data ports are described using five stardard parts. The message's Descriptor is sent to the target data port on a message sideband bus, and the other four 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) 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 Media Block Write message may have End of Thread set for pixel shader threads dispatched by the windower in contiguous dispatch mode.

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

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:

Description of Message Summary Tables

Column Heading	Description
Addr Align	Specifies the required address alignment of the operation. This can be byte (B), DWord (DW), QWord (QW), OWord (OW), or HWord (HW). In most cases, the address must be aligned for the data width of the operation.
Data Width	Specifies the width of the data operation. This can be byte (B), DWord (DW), QWord (QW), OWord (OW), or HWord (OW).
R/W	Specifies if the operation is a Read or Write (R or W). Atomic operations that return data are classified here as reads, and ones that don't are classified here as writes.
Address Model	Specifies the address model used by this message operation: BTS, SLM, A32, or A64. Read-only data port accesses to the constant cache or sampler cache are labeled "CC BTS" or "SC BTS".
Surface Type	Specifies the supported surface type. See the Surfaces Types section of this volume.
SIMD Slots	In the first table, this summarizes all the SIMD slot configurations supported for this operation and address model. In the second table, this specifies the specific SIMD slot configuration supported by the specific message operation.
Data Elements	Specifies all the sizes of data elements supported by the message operation. For blocks, this is the number of blocks. For surfaces, this is the list of channels that are enabled in the message descriptor. See MDC_CMASK Channel Mask Message Descriptor Control Field in this volume.

Column Heading	Description
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 heder to provide additional parameters to control their operation, in addition to the parameters in the message descriptor. Each section's table lists which message headers are legal to use. See Message Headers for a full list of the message headers that are supported by all the messages.
Address Payload	Most messages perform SIMD operations using address offsets for the operation from the address payload. See Message Address Payloads for a full list of the message address payloads that are supported by all the messages. Each section's table lists which message headers are legal to use.
Source Payload	Write messages and atomic messages generally provide data in a source data payload. See Message Data Payloads for a full list of the message data payloads that are supported by all the messages. Each section's table lists which message source payloads are legal to use.
Writeback Payload	Read messages generally return data in a writeback data payload. See Message Data Payloads for a full list of the message data payloads that are supported by all the messages. Each section's table lists which message writeback payloads are used.

Messages

The message operations on data ports are described this section. The operations are organized into subsections by their addressing operation group (blocks, scattered, surfaces, atomic operations), and then by their data width and type. Each subsection describes the messages that have common semantics and limitations. (For example, read and write operations are described in the same subsection.)

Summary of Message Groups

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
Scattered and Block R/W					1, 2	1, 2, 4, 8		
Byte Scattered R/W	B	B	BTS	BUFFER, NULL	8, 16	1, 2, 4	Surface	SMBLK
	B	B	SLM	BUFFER	8, 16	1, 2, 4	Wrap	SMBLK
	B	B	A32	BUFFER	8, 16	1, 2, 4	PTSS	SMBLK
	B	B	A32	BUFFER	8, 16	1, 2, 4	GenState	SMBLK
	B	B	A64	BUFFER	8, 16	1, 2, 4	Canonical	SMBLK
Byte Scaled R/W	B	B	SLM	BUFFER	8, 16	1, 2, 4	Wrap	SMBLK
	B	B	A32	BUFFER	8, 16	1, 2, 4	GenState	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
DW Scaled R/W	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
QW Scaled 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	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
OW Dual Block R/W	OW	OW	BTS	BUFFER, NULL	2	1, 4	Surface	DBLK
	OW	OW	A32	BUFFER	2	1, 4	PTSS	DBLK
	OW	OW	A64	BUFFER	2	1, 4	Canonical	DBLK
HW Block R/W	HW	HW	A32	BUFFER	1	1, 2, 4, 8	PTSS	BLKCM
HW	HW	A64	BUFFER	1	1, 2, 4, 8	Canonical	BLKCM	
Surface R/W					2, 8, 16	1		
Scattered Untyped Surface R/W	DW	OW	BTS	BUFFER, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	BUFFER, NULL	8, 16	{Chan 1-4}	Surface	SMPSM
	DW	OW	SLM	BUFFER	4x2	1	Wrap	SM4x2
	DW	DW	SLM	BUFFER	8, 16	{Chan 1-4}	Wrap	SMPSM
	DW	OW	A32	BUFFER	4x2	1	GenState	SM4x2

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
	DW	DW	A32	BUFFER	8, 16	{Chan 1-4}	GenState	SMPSM
	DW	OW	A64	BUFFER	4x2	1	Canonical	SM4x2
	DW	DW	A64	BUFFER	8, 16	{Chan 1-4}	Canonical	SM
Scaled Untyped Surface R/W	DW	OW	BTS	STRBUF, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	STRBUF, NULL	8, 16	{Chan 1-4}	Surface	SMPSM
	DW	OW	SLM	BUFFER	4x2	1	Wrap	SM4x2
	DW	DW	SLM	BUFFER	8, 16	{Chan 1-4}	Wrap	SM
	DW	OW	A32	BUFFER	4x2	1	GenState	SM4x2
	DW	DW	A32	BUFFER	8, 16	{Chan 1-4}	GenState	SMPSM
	DW	OW	A64	BUFFER	4x2	1	Canonical	SM4x2
	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	4x2	1	Surface	SM4x2
	B	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	Surface	SGPSM
Scattered MSAA Typed Surface R/W	B	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	4x2	1	Surface	SM4x2
	B	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	Surface	SGPSM
Atomic Operations					2, 8, 16	1		
DW Untyped Atomic Integer	DW	DW	BTS	BUFFER, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	BUFFER, NULL	8, 16	1	Surface	SMPSM
	DW	DW	SLM	BUFFER	4x2	1	Wrap	SM4x2
	DW	DW	SLM	BUFFER	8, 16	1	Wrap	SMPSM
	DW	DW	A32	BUFFER	4x2	1	GenState	SM4x2
	DW	DW	A32	BUFFER	8, 16	1	GenState	SMPSM
	DW	DW	A64	BUFFER	4x2	1	Canonical	SM4x2
	DW	DW	A64	BUFFER	8	1	Canonical	SM
DW Scaled Untyped Atomic Integer	DW	DW	BTS	STRBUF, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	STRBUF, NULL	8, 16	1	Surface	SMPSM
QW Untyped Atomic Integer	QW	QW	BTS	BUFFER, NULL	4x2	1	Surface	SM4x2
	QW	QW	BTS	BUFFER, NULL	8, 16	1	Surface	SMPSM

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
	QW	QW	A32	BUFFER	4x2	1	GenState	SM4x2
	QW	QW	A32	BUFFER	8, 16	1	GenState	SMPSM
	QW	QW	A64	BUFFER	4x2	1	Canonical	SM4x2
	QW	QW	A64	BUFFER	8	1	Canonical	SM
QW Scaled Untyped Atomic Integer	QW	QW	BTS	STRBUF, NULL	4x2	1	Surface	SM4x2
	QW	QW	BTS	STRBUF, NULL	8, 16	1	Surface	SMPSM
OW Untyped Atomic Integer	OW	OW	A64	BUFFER	4x2	1	Canonical	SM4x2
	OW	OW	A64	BUFFER	8	1	Canonical	SM
DW Untyped Atomic Float	DW	DW	BTS	BUFFER, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	BUFFER, NULL	8, 16	1	Surface	SMPSM
	DW	DW	SLM	BUFFER	4x2	1	Wrap	SM4x2
	DW	DW	SLM	BUFFER	8, 16	1	Wrap	SMPSM
	DW	DW	A32	BUFFER	4x2	1	GenState	SM4x2
	DW	DW	A32	BUFFER	8, 16	1	GenState	SMPSM
	DW	DW	A64	BUFFER	4x2	1	Canonical	SM4x2
	DW	DW	A64	BUFFER	8	1	Canonical	SM
DW Scaled Untyped Atomic Float	DW	DW	BTS	STRBUF, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	STRBUF, NULL	8, 16	1	Surface	SMPSM
DW Typed Atomic Integer	DW	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	4x2	1	Surface	SM4x2
	DW	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	Surface	SGPSM
DW Atomic Counter	DW	DW	BTS	Any	4x2	1	Ignored	SM4x2
	DW	DW	BTS	Any	8	1	Ignored	SGPSM, SG
Media Block R/W								
Transpose Read	DW	DW	BTS	2D, NULL	1, 2, 4, 8	{1, 2, 4, 8}	Media	Ignored
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]	Wrap	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	A32	BUFFER	8, 16	1, 2, 4	(Base + (Offset{Slot})→B[Elem]	GenState	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.

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

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	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	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	A32	BUFFER	SIMD8	{1, 2, 4}	MSD2R_BS	{Forbidden}	MAP32b_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{1, 2, 4}	MSD2W_BS	{Forbidden}	MAP32b_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	{1, 2, 4}	MSD2R_BS	{Forbidden}	MAP32b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{1, 2, 4}	MSD2W_BS	{Forbidden}	MAP32b_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1R_A6	{Forbidde	MAP64b_SI	{Forbidden}	{1, 1, 1}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
					4_BS	n}	MD8		MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1W_A64_BS	{Forbidden}	MAP64b_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2R_A64_BS	{Forbidden}	MAP64b_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2W_A64_BS	{Forbidden}	MAP64b_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1R_A64_BS	{Forbidden}	MAP64b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1W_A64_BS	{Forbidden}	MAP64b_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2R_A64_BS	{Forbidden}	MAP64b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2W_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 of these cases a different message may provide higher performance.	

Programming Note	
Context:	Byte Scattered ReadWrite Messages
For byte scattered read and write the buffer size must be a multiple of 4 bytes.	

Byte Scaled 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	B	R/W	SLM	BUFFER	8, 16	1, 2, 4	(Base + Sideband + (Offset{Slot} * (Pitch+1)))→B[Elem]	Wrap	SMBLK
B	B	R/W	A32	BUFFER	8, 16	1, 2, 4	(Base + Buffer + Sideband + (Offset{Slot} * (Pitch+1)))→B[Elem]	GenState	SMBLK
B	B	R/W	A64	BUFFER	8, 16	1, 2, 4	(0 + Buffer + Sideband + (Offset{Slot} * (Pitch+1)))→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 scaled by the **Pitch**. The Global Offset is added to each of the specific offsets when the message header is present. The surface format is ignored, data is returned from the buffer to the GRF without format conversion.

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	SLM	BUFFER	SIMD8	{1, 2, 4}	MSD2R_BS	{Forbidden}	MAP32b_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{1, 2, 4}	MSD2W_BS	{Forbidden}	MAP32b_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD2R_BS	{Forbidden}	MAP32b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD2W_BS	{Forbidden}	MAP32b_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	{1, 2, 4}	MSD2R_BS	MH2_A32	MAP32b_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{1, 2, 4}	MSD2W_BS	MH2_A32	MAP32b_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	A32	BUFFER	SIMD16	{1, 2, 4}	MSD2R_BS	MH2_A32	MAP32b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{1, 2, 4}	MSD2W_BS	MH2_A32	MAP32b_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2R_A64_BS	MH2_A64	MAP32b_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2W_A64_BS	MH2_A64	MAP32b_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2R_A64_BS	MH2_A64	MAP32b_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2W_A64_BS	MH2_A64	MAP32b_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}

Programming Note	
Context:	Byte Scaled Read/Write Messages
For byte scattered read and write 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, 8	(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.

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,NULL	SIMD8	1	MSD0R_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD0W_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD0R_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD0W_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD0R_DWS	{Opt} MH_A32_GO	MAP32b_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD0W_DWS	{Opt} MH_A32_GO	MAP32b_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD0R_DWS	{Opt} MH_A32_GO	MAP32b_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD0W_DWS	{Opt} MH_A32_GO	MAP32b_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4, 8}	MSD1R_A64_DWS	{Forbidden}	MAP64b_SIMD8	{Forbidden}	{1, 2, 4, 8} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4, 8}	MSD1W_A64_DWS	{Forbidden}	MAP64b_SIMD8	{1, 2, 4, 8} 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	SIMD 8	{1, 2, 4, 8}	MSD2R_A64_DWS	{Forbidden}	MAP64b_SIMD8	{Forbidden}	{1, 2, 4, 8} MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	{1, 2, 4, 8}	MSD2W_A64_DWS	{Forbidden}	MAP64b_SIMD8	{1, 2, 4, 8} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD 16	{1, 2, 4}	MSD1R_A64_DWS	{Forbidden}	MAP64b_SIMD16	{Forbidden}	{1, 2, 4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD 16	{1, 2, 4}	MSD1W_A64_DWS	{Forbidden}	MAP64b_SIMD16	{1, 2, 4} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD 16	{1, 2, 4}	MSD2R_A64_DWS	{Forbidden}	MAP64b_SIMD16	{Forbidden}	{1, 2, 4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD 16	{1, 2, 4}	MSD2W_A64_DWS	{Forbidden}	MAP64b_SIMD16	{1, 2, 4} MDP_DW_SIMD16	{Forbidden}
R	CC BTS	BUFFER,NULL	SIMD 8	1	MSD_CC_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD8	{Forbidden}	MDP_DW_SIMD8
R	CC BTS	BUFFER,NULL	SIMD 16	1	MSD_CC_DWS	{Opt} MH_BTS_GO	MAP32b_SIMD16	{Forbidden}	MDP_DW_SIMD16

Programming Note	
Context:	DWord Scattered Read/Write Messages
For BTS and A32 address models, the offsets in the address payload must in the range 0 - 3FFFFFFFh. If the upper 2 bits are non-zero, the result is undefined.	

DWord Scaled 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	A64	BUFFER	8, 16	1, 2, 4, 8	(0+Buffer+Sideband+(Offset{Slot}*(Pitch+1)))→DW[Elem]	Canonical	SMBLK

This message reads or writes 8 or 16 scattered DWords starting at each offset. The offsets are scaled by the **Pitch**.

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, 2, 4, 8}	MSD2R_A64_DWS	MH2_A64	MAP32b_SIMD8	{Forbidden}	{1, 2, 4, 8} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4, 8}	MSD2W_A64_DWS	MH2_A64	MAP32b_SIMD8	{1, 2, 4, 8} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2R_A64_DWS	MH2_A64	MAP32b_SIMD16	{Forbidden}	{1, 2, 4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD2W_A64_DWS	MH2_A64	MAP32b_SIMD16	{1, 2, 4} MDP_DW_SIMD16	{Forbidden}

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 takes a set of QWord-aligned offsets, and reads or writes SIMD8 scattered QWords starting at each offset.

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 reads where the indices differ (read one channel per message).

SIMD8/16 writes where the indices differ (write one channel per message).

General purpose QWord 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	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/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}	MSD2R_A64_QWS	{Forbidden}	MAP64b_SIMD8	{Forbidden}	{1, 2, 4} MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2W_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}
R	A64	BUFFER	SIMD16	{1, 2}	MSD2R_A64_QWS	{Forbidden}	MAP64b_SIMD16	{Forbidden}	{1, 2} MDP_QW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2}	MSD2W_A64_QWS	{Forbidden}	MAP64b_SIMD16	{1, 2} MDP_QW_SIMD16	{Forbidden}

QWord Scaled 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+Buffer+Sideband+(Offset{Slot}*(Pitch+1)))→QW[Elem]	Canonical	SMBLK

This message takes a set of QWord-aligned offsets, and reads or writes SIMD8 scattered QWords starting at each offset. The offsets are scaled by the **Pitch**.

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, 2, 4}	MSD2R_A64_QWS	MH2_A64	MAP32b_SIMD8	{Forbidden}	{1, 2, 4} MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD2W_A64_QWS	MH2_A64	MAP32b_SIMD8	{1, 2, 4} MDP_QW_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	SIMD 16	{1, 2}	MSD2R_A64_QWS	MH2_A64	MAP32b_SIMD16	{Forbidden}	{1, 2} MDP_QW_SIMD16
W	A64	BUFFER	SIMD 16	{1, 2}	MSD2W_A64_QWS	MH2_A64	MAP32b_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	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. The pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

The execution mask bits may disable accesses on the corresponding blocks. 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 where the index for each pixel/vertex is the same.
- Block constant reads, scratch memory 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	BUFFER,NULL	1	{1, 2, 4, 8}	MSD0R_OWB	MH_BTS_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B

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	1	{1, 2, 4, 8}	MSD0W_OWB	MH_BTS_GO	{Forbidden}	{1, 1, 2, 4} MDP_OW_B	{Forbidden}
R	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWB	MH_A32_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B
W	A32	BUFFER	1	{1, 2, 4, 8}	MSD0W_OWB	MH_A32_GO	{Forbidden}	{1, 1, 2, 4} MDP_OW_B	{Forbidden}
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_OWB	MH_A64_OWB	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B
W	A64	BUFFER	1	{1, 2, 4, 8}	MSD1W_A64_OWB	MH_A64_OWB	{Forbidden}	{1, 1, 2, 4} MDP_OW_B	{Forbidden}
R	CC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_CC_OWB	MH_BTS_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B

For the one-constant case, either the high or low half of the payload register is used depending on the half selected in Block Size, and the other half of the payload register is ignored.

Programming Note	
Context:	OWord Block Read/Write Messages
For the BTS and A32 address models, the offsets in the address payload must be in the range 0 - 0FFFFFFFh. If the upper 4 bits are non-zero, the result is undefined.	

OWord Aligned Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	OW	R	BTS	BUFFER, NULL	1	1, 2, 4, 8	(Base+GlobalOffset)→OW[Elem]	Surface	Ignored
DW	OW	R	A32	BUFFER	1	1, 2, 4, 8	(Base+Buffer+GlobalOffset)→OW[Elem]	PTSS	Ignored
DW	OW	R	A64	BUFFER	1	1, 2, 4, 8	(0+BlockOffset0)→OW[Elem]	Canonical	Ignored

Functional Description
This message takes one DWord-aligned offset (Global Offset), and reads 1, 2, 4, or 8 contiguous OWords starting at that offset.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format. The pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

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}	{1, 1, 2, 4} MDP_OW_B
R	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWAB	MH_A32_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_OWAB	MH_A64_OWB	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B
R	CC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_CC_OWAB	MH_BTS_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B
R	SC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_SC_OWAB	MH_BTS_GO	{Forbidden}	{Forbidden}	{1, 1, 2, 4} MDP_OW_B

Programming Note
For read/write cache, only the read path supports the unaligned OWord Block access.

OWord Dual 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	2	1, 4	Base→OW[GlobalOffset+Offset{0, 1}+Elem]	Surface	DBLK
OW	OW	R / W	A32	BUFFER	2	1, 4	(Base+Buffer)→OW[GlobalOffset+Offset{0, 1}+Elem]	PTSS	DBLK
OW	OW	R / W	A64	BUFFER	2	1, 4	(0+BlockOffset{0, 1})→OW[Elem]	Canonical	DBLK

This message takes two offsets, and reads or writes 1 or 4 contiguous OWords starting at each offset. The Global Offset is added to each of the specific offsets.

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.

The execution mask bits may disable accesses on the corresponding blocks. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Applications:

- SIMD4x2 constant buffer reads where the indices of each vertex/pixel differ (if two indices are the same, hardware optimizes the cache accesses and does only one cache access).
- SIMD4x2 scratch space reads/writes where the indices differ.

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	2	{1, 4}	MSD0R_OWDB	{Opt} MH_BTS_GO	MAP32b_SIMD4x2	{Forbidden}	{1, 4} MDP_OW_DB
W	BTS	BUFFER, NULL	2	{1, 4}	MSD0W_OWDB	{Opt} MH_BTS_GO	MAP32b_SIMD4x2	{1, 4} MDP_OW_DB	{Forbidden}
R	A32	BUFFER	2	{1, 4}	MSD0R_OWDB	{Opt} MH_A32_GO	MAP32b_SIMD4x2	{Forbidden}	{1, 4} MDP_OW_DB
W	A32	BUFFER	2	{1, 4}	MSD0W_OWDB	{Opt} MH_A32_GO	MAP32b_SIMD4x2	{1, 4} MDP_OW_DB	{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	2	{1, 4}	MSD1R_A64_OWDB	MH_A64_OWDB	{Forbidden}	{Forbidden}	{1, 4} MDP_OW_DB
W	A64	BUFFER	2	{1, 4}	MSD1W_A64_OWDB	MH_A64_OWDB	{Forbidden}	{1, 4} MDP_OW_DB	{Forbidden}
R	CC BTS	BUFFER, NULL	2	{1, 4}	MSD_CC__OWDB	{Opt} MH_BTS_GO	MAP32b_SIMD4x2	{Forbidden}	{1, 4} MDP_OW_DB

Programming Note	
Context:	OWord Dual Block Read/Write Messages
For BTS and A32 address models, the offsets in the address payload must in the range 0 - 0FFFFFFFh. If the upper 4 bits are non-zero, the result is undefined.	

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 execution mask bits may disable accesses on the corresponding blocks. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

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}	{1, 2, 4, 8} MDP_HW_B
W	A32	BUFFER	1	{1, 2, 4, 8}	MSD0W_SB	MH_A32_HWB	{Forbidden}	{1, 2, 4, 8} MDP_HW_B	{Forbidden}

Channel Mode Interpretation
Two modes of channel-enable interpretation are provided: • DWord, which support a SIMD8 or SIMD16 DWord channel-serial view of a register, and • OWord, which supports a SIMD4x2 view of a register.

Programming Note	
Context:	HWord Block ReadWrite Messages
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|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 from the A32 general state base address.	

Programming Note
Although Scratch Block messages are always used only inside a thread's local scratch space, they are assumed to be coherent memory accesses (MDC_STATELESS A32_A64).

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
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. The pitch is 1 byte 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.

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 Hword index.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_HWB	MH_A64_HWB	{Forbidden}	{Forbidden}	{1, 2, 4, 8} MDP_HW_B
W	A64	BUFFER	1	{1, 2, 4, 8}	MSD1W_A64_HWB	MH_A64_HWB	{Forbidden}	{1, 1, 2, 4} MDP_HW_B	{Forbidden}

Channel Mode Interpretation
Two modes of channel-enable interpretation are provided: • DWord, which support a SIMD8 or SIMD16 DWord channel-serial view of a register, and • OWord, which supports a SIMD4x2 view of a register.

Programming Note
Although Scratch Block messages are only used inside a thread's local scratch space, they are assumed to be coherent memory accesses (MDC_STATELESS A32_A64).

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_STRBUF (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	OW	R/W	BTS	BUFFER,NULL	4x2	1	(Base+U{Slot})→DW[Chan]	Surface	SM4x2
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	{Chan 1-4}	(Base+U{Slot})→DW[Chan]	Surface	SMPSM
DW	OW	R/W	SLM	BUFFER	4x2	1	(Base+U{Slot})→DW[Chan]	Wrap	SM4x2
DW	DW	R/W	SLM	BUFFER	8, 16	{Chan 1-4}	(Base+U{Slot})→DW[Chan]	Wrap	SMPSM

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	OW	R/W	A32	BUFFER	4x2	1	(Base+Buffer+U{Slot})→DW[Chan]	GenState	SM4x2
DW	DW	R/W	A32	BUFFER	8, 16	{Chan 1-4}	(Base+Buffer+U{Slot})→DW[Chan]	GenState	SMPSM
DW	OW	R/W	A64	BUFFER	4x2	1	(0+U{Slot})→DW[Chan]	Canonical	SM4x2
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. 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,NULL	SIMD4x2	1	MSD1R_US	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	BTS	BUFFER,NULL	SIMD4x2	1	MSD1W_US	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	{Chan 1-4}	MSD1R_US	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	{Chan 1-4}	MSD1W_US	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	{Chan 1-4}	MSD1R_US	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	{Chan 1-4}	MSD1W_US	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD4x2	1	MSD1R_US	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	SLM	BUFFER	SIMD4x2	1	MSD1W_US	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	SLM	BUFFER	SIMD4x2	1	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	SLM	BUFFER	SIMD4x2	1	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1R_US	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD4x2	1	MSD1R_US	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A32	BUFFER	SIMD4x2	1	MSD1W_US	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	A32	BUFFER	SIMD4x2	1	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A32	BUFFER	SIMD4x2	1	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{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	{Chan 1-4}	MSD1R_US	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD8	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD16	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD1R_A64_US	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A64	BUFFER	SIMD4x2	1	MSD1W_A64_US	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD2R_A64_US	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A64	BUFFER	SIMD4x2	1	MSD2W_A64_US	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_DW_SIMD4x2	{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/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A64	BUFFER	SIMD8	{Chan 1-4}	MSD2R_A64_US	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{Chan 1-4}	MSD2W_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}
R	A64	BUFFER	SIMD16	{Chan 1-4}	MSD2R_A64_US	{Forbidden}	MAP64b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{Chan 1-4}	MSD2W_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.

Scaled Untyped 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
DW	OW	R/W	BTS	STRBUF,NULL	4x2	1	(Base+(U{Slot}*Pitch)+V{Slot})→DW[Chan]	Surface	SM4x2
DW	DW	R/W	BTS	STRBUF,NULL	8, 16	{Chan 1-4}	(Base+(U{Slot}*Pitch)+V{Slot})→DW[Chan]	Surface	SMPSM
DW	OW	R/W	SLM	BUFFER	4x2	1	(Base+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	Wrap	SM4x2
DW	DW	R/W	SLM	BUFFER	8, 16	{Chan 1-4}	(Base+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	Wrap	SM
DW	OW	R/W	A32	BUFFER	4x2	1	(Base+Buffer+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	GenState	SM4x2
DW	DW	R/W	A32	BUFFER	8, 16	{Chan 1-4}	(Base+Buffer+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	GenState	SMPSM

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	OW	R/W	A64	BUFFER	4x2	1	(0+Buffer+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	Canonical	SM4x2
DW	DW	R/W	A64	BUFFER	8, 16	{Chan 1-4}	(0+Buffer+Sideband+(U{Slot}*(Pitch+1)))→DW[Chan]	Canonical	SM

This message allows scaled read/write accesses to untyped scaled surfaces using 8 or 16 offsets. Up to 4 DWords are accessed beginning at the byte address determined by the U offset and the scaling pitch. These 4 DWords correspond to the red, green, blue, and alpha channels, in that order and with red mapping to the lowest order DWord. After scaling calculation, the 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	STRBUF,NULL	SIMD 4x2	1	MSD1R_US	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	BTS	STRBUF,NULL	SIMD 4x2	1	MSD1W_US	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD 8	{Chan 1-4}	MSD1R_US	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD 8	{Chan 1-4}	MSD1R_US	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	BTS	STRBUF,NULL	SIMD 16	{Chan 1-4}	MSD1W_US	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	BTS	STRBUF,NULL	SIMD 16	{Chan 1-4}	MSD1W_US	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD 4x2	1	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	SLM	BUFFER	SIMD 4x2	1	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	SLM	BUFFER	SIMD 8	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	SLM	BUFFER	SIMD 8	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD 16	{Chan 1-4}	MSD2R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD 16	{Chan 1-4}	MSD2W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD 4x2	1	MSD2R_US	MH2_A32	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A32	BUFFER	SIMD 4x2	1	MSD2W_US	MH2_A32	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	A32	BUFFER	SIMD 8	{Chan 1-4}	MSD2R_US	MH2_A32_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A32	BUFFER	SIMD 8	{Chan 1-4}	MSD2W_US	MH2_A32_PSM	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD 16	{Chan 1-4}	MSD2R_US	MH2_A32_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A32	BUFFER	SIMD 16	{Chan 1-4}	MSD2W_US	MH2_A32_PSM	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD2R_A64_US	MH2_A64	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	A64	BUFFER	SIMD 4x2	1	MSD2W_A64_US	MH2_A64	MAP32b_USU_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	A64	BUFFER	SIMD 8	{Chan 1-4}	MSD2R_A64_US	MH2_A64	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	{Chan 1-4}	MSD2W_A64_US	MH2_A64	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD 16	{Chan 1-4}	MSD2R_A64_US	MH2_A64	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	A64	BUFFER	SIMD 16	{Chan 1-4}	MSD2W_A64_US	MH2_A64	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}

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	4x2	1	(Surface[U, V, R, LOD])→DW[Chan]	Surface	SM4x2
B	DW	R/W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	(Surface[U, V, R, LOD])→DW[Chan]	Surface	SGPSM

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 4x2	1	MSD1R_TS	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	{Forbidden}	MDP_DW_SIMD 4x2
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD 4x2	1	MSD1W_TS	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	MDP_DW_SIMD 4x2	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD 8	{Chan 1-4}	MSD1R_TS	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	{Chan 1-4}	MSD1W_TS	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	1	MSD1R_TS	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	{Forbidden}	MDP_TileW_SIMD8
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	1	MSD1W_TS	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	MDP_TileW_SIMD8	{Forbidden}

The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled.

The **Channel Mask** in the message descriptor identifies whether the corresponding channel access is disabled. The surface state **Shader Channel Select** fields identify whether the channels are swizzled for the access.

See Surface Formats for the description of how the data is formatted for Typed Surface read and write operations.

MSAA Typed Surface ReadWrite 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	2D, NULL	4x2	1	(Surface[U, V, R, LOD, MSAA])→DW[Chan]	Surface	SM4x2
B	DW	R/W	BTS	2D, NULL	8	1	(Surface[U, V, R, LOD, MSAA])→DW[Chan]	Surface	SGPSM

This message allows direct read/write accesses of multi-sample 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 2D surface can be either a normal 2D surface, a 2D Array surface, or a 2D mip-mapped surface. Sample planes are laid out sequentially for each array element or mip-mapped surface.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. If the MSAA sample number is larger than **Number of Multisamples** in the surface state, the access is out of bounds.

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	SIMD 4x2	1	MSD1R_TS	{Opt} MH_IGNORE	MAP32b_MSAA_TS_SIMD4x2	{Forbidden}	MDP_DW_SIMD4x2
W	BTS	2D, NULL	SIMD 4x2	1	MSD1W_TS	{Opt} MH_IGNORE	MAP32b_MSAA_TS_SIMD4x2	MDP_DW_SIMD4x2	{Forbidden}
R	BTS	2D, NULL	SIMD 8	{Chan 1-4}	MSD1R_TS	{Opt} MH1_BTS_PSM	MAP32b_MSAA_TS_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	BTS	2D, NULL	SIMD 8	{Chan 1-4}	MSD1W_TS	{Opt} MH1_BTS_PSM	MAP32b_MSAA_TS_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}

The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled.

The **Channel Mask** in the message descriptor identifies whether the corresponding channel access is disabled. The surface state **Shader Channel Select** fields identify whether the channels are swizzled for the access.

See Surface Formats for the description of how the data is formatted for Typed Surface read and write operations.

Programming Note	
Context:	MSAA Typed Surface ReadWrite Messages
This message only supports PLANAR multi-sample surfaces (surface state **Multisample Surface Storage Format** MSS), without a color control auxillary surface (surface state **Auxillary Surface Mode** NONE) and without color compression (surface state **Render Target Compression Enable** FALSE).	

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 STRBUF, 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, each shader channel select in the surface state must be set to the same surface channel (R = SCS_RED, G = SCS_GREEN, B = SCS_BLUE, A = SCS_ALPHA). Only the red surface channel is used (the first DWord).

The MDC_AOP and MDC_FOP are specified in the Atomic Operation Message Descriptor Control Fields section.

The new value of the destination is computed based on the old value of the destination, and up to two additional sources included in the message (src0 and src1). The sources used by the specific operations are described in the **MDC_AOP** and **MDC_FOP** tables. The AOP operations are subdivided into unary, binary, and trinary operand groupings. Source data payload are only sent for the binary and trinary operations.

The double-width AOP operation CMPWR_2W, also known as CMPRW8B on DWords and CMPWR16B on QWords, require the read-modify-write address offsets to be naturally aligned (QWord and OWord respectively).

The **MDC_RDC** message-specific control field controls whether a value is returned by the atomic message. The value returned depends on the message specific operation (see **MDC_AOP** and **MDC_FOP**).

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	4x2	1	(Base+U{Slot})→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})→DW[0]	Surface	SMPSM
DW	DW	R/W	SLM	BUFFER	4x2	1	(Base+U{Slot})→DW[0]	Wrap	SM4x2
DW	DW	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})→DW[0]	Wrap	SMPSM
DW	DW	R/W	A32	BUFFER	4x2	1	(Base+Buffer+U{Slot})→DW[0]	GenState	SM4x2
DW	DW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})→DW[0]	GenState	SMPSM
DW	DW	R/W	A64	BUFFER	4x2	1	(0+U{Slot})→DW[0]	Canonical	SM4x2
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. 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	SIMD4x2	1	MSD1R_DWAI1_4x2	{Opt}MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	BTS	BUFFER,NULL	SIMD4x2	1	MSD1W_DWAI1_4x2	{Opt}MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD4x2	1	MSD1R_DWAI2_4x2	{Opt}MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	BUFFER,NULL	SIMD4x2	1	MSD1W_DWAI2_4x2	{Opt}MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD15	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD15	{Forbidden}
R	BTS	BUFFER,NULL	SIMD4x2	1	MSD1R_DWAI3_4x2	{Opt}MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1

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	SIMD 4x2	1	MSD1W_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD 8	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD 8	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD 16	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD 16	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	SLM	BUFFER	SIMD 4x2	1	MSD1R_DWAI1_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	SLM	BUFFER	SIMD 4x2	1	MSD1W_DWAI1_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD 8	1	MSD1R_DWAI1	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD 8	1	MSD1W_DWAI1	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD 16	1	MSD1R_DWAI1	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD 16	1	MSD1W_DWAI1	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD 4x2	1	MSD1R_DWAI2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	SLM	BUFFER	SIMD 4x2	1	MSD1W_DWAI2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAI2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAI2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD4x2	1	MSD1R_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	SLM	BUFFER	SIMD4x2	1	MSD1W_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAI3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAI3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A32	BUFFER	SIMD4x2	1	MSD1R_DWAI1_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	A32	BUFFER	SIMD4x2	1	MSD1W_DWAI1_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI1	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI1	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	{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	SIMD16	1	MSD1R_DWAI1	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI1	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD4x2	1	MSD1R_DWAI2_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	A32	BUFFER	SIMD4x2	1	MSD1W_DWAI2_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD4x2	1	MSD1R_DWAI3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	A32	BUFFER	SIMD4x2	1	MSD1W_DWAI3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD1R_A64_DWAI1_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_DWAI1_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD 8	1	MSD1R_A64_DWAI1	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	1	MSD1W_A64_DWAI1	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD1R_A64_DWAI2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_DWAI2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	A64	BUFFER	SIMD 8	1	MSD1R_A64_DWAI2	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	1	MSD1W_A64_DWAI2	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD1R_A64_DWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_DWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	A64	BUFFER	SIMD 8	1	MSD1R_A64_DWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	1	MSD1W_A64_DWAI3	{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.	

DWord Scaled 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	STRBUF,NULL	4x2	1	(Base+(U{Slot}*Pitch)+V{Slot})→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	STRBUF,NULL	8, 16	1	(Base+(U{Slot}*Pitch)+V{Slot})→DW[0]	Surface	SMPSM

This message performs atomic integer operations on untyped surfaces using 8 or 16 offsets. One DWord is accessed beginning at the byte address determined. Each U offset is scaled by Pitch. Both the Surface Pitch and V 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	STRBUF,NULL	SIMD4x2	1	MSD1R_DWAI1_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	BTS	STRBUF,NULL	SIMD4x2	1	MSD1W_DWAI1_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	{Forbidden}	{Forbidden}
R	BTS	STRBUF,NULL	SIMD8	1	MSD1R_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD8	1	MSD1W_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	{Forbidden}	{Forbidden}
R	BTS	STRBUF,NULL	SIMD16	1	MSD1R_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	STRBUF,NULL	SIMD16	1	MSD1W_DWAI1	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD16	{Forbidden}	{Forbidden}
R	BTS	STRBUF,NULL	SIMD4x2	1	MSD1R_DWAI2_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	STRBUF,NULL	SIMD4x2	1	MSD1W_DWAI2_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	STRBUF,NULL	SIMD8	1	MSD1R_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD8	1	MSD1W_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	STRBUF,NULL	SIMD16	1	MSD1R_DWAI2	{Opt}MH1_BTS_PSM	MAP32b_USUV_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
W	BTS	STRBUF,NULL	SIMD16	1	MSD1W_DWAI2	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	STRBUF,NULL	SIMD4x2	1	MSD1R_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	BTS	STRBUF,NULL	SIMD4x2	1	MSD1W_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD8	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD8	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD16	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	STRBUF,NULL	SIMD16	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_DW2	{Forbidden}

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	BTS	BUFFER,NULL	4x2	1	(Base+U{Slot})→QW[0]	Surface	SM4x2
QW	QW	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})→QW[0]	Surface	SMPSM
QW	QW	R/W	A32	BUFFER	4x2	1	(Base+Buffer+U{Slot})→QW[0]	GenState	SM4x2
QW	QW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})→QW[0]	GenState	SMPSM
QW	QW	R/W	A64	BUFFER	4x2	1	(0+U{Slot})→QW[0]	Canonical	SM4x2
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 and the surface type is SURFTYPE_BUFFER. One QWord is accessed beginning at the byte address determined. 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	BTS	BUFFER,NULL	SIMD 4x2	1	MSD1R_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_QW2	MDP_AOP4x2_QW1
W	BTS	BUFFER,NULL	SIMD 4x2	1	MSD1W_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_QW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD 8	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_QW2	MDP_AOP8_QW1
W	BTS	BUFFER,NULL	SIMD 8	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_QW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD 16	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_QW2	MDP_AOP16_QW1
W	BTS	BUFFER,NULL	SIMD 16	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_QW2	{Forbidden}
R	A32	BUFFER	SIMD 4x2	1	MSD1R_DWAI3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_QW2	MDP_AOP4x2_QW1
W	A32	BUFFER	SIMD 4x2	1	MSD1W_DWAI3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_QW2	{Forbidden}
R	A32	BUFFER	SIMD 8	1	MSD1R_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_QW2	MDP_AOP8_QW1
W	A32	BUFFER	SIMD 8	1	MSD1W_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_QW2	{Forbidden}
R	A32	BUFFER	SIMD 16	1	MSD1R_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_QW2	MDP_AOP16_QW1
W	A32	BUFFER	SIMD 16	1	MSD1W_DWAI3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_QW2	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD1R_A64_QWAI1_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_QW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_QWAI1_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	{Forbidden}	{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	SIMD8	1	MSD1R_A64_QWAI1	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI1	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD1R_A64_QWAI2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_QW1	MDP_AOP4x2_QW1
W	A64	BUFFER	SIMD4x2	1	MSD1W_A64_QWAI2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_QW1	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_QWAI2	{Forbidden}	MAP64b_USU_SIMD8	MDP_QW_SIMD8	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI2	{Forbidden}	MAP64b_USU_SIMD8	MDP_QW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD1R_A64_QWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_QW2	MDP_AOP4x2_QW1
W	A64	BUFFER	SIMD4x2	1	MSD1W_A64_QWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_QW2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_QWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_QW2	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_QW2	{Forbidden}
R	A64	BUFFER	SIMD4x2	1	MSD1R_A64_DWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_QW2	MDP_AOP4x2_QW1
W	A64	BUFFER	SIMD4x2	1	MSD1W_A64_DWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_QW2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_QW2	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAI3	{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.	

QWord Scaled 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	BTS	STRBUF,NULL	4x2	1	(Base+(U{Slot}*Pitch)+V{Slot})→QW[0]	Surface	SM4x2
QW	QW	R/W	BTS	STRBUF,NULL	8, 16	1	(Base+(U{Slot}*Pitch)+V{Slot})→QW[0]	Surface	SMPSM

This message performs QWord atomic integer operations on untyped surfaces using 8 or 16 offsets. The surface format is RAW and the surface type is SURFTYPE_STRBUF. One QWord is accessed beginning at the byte address determined. Each U offset is scaled by Pitch. Both the Surface Pitch and V 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	BTS	STRBUF,NULL	SIMD 4x2	1	MSD1R_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_QW2	MDP_AOP4x2_QW1
W	BTS	STRBUF,NULL	SIMD 4x2	1	MSD1W_DWAI3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_QW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD 8	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_QW2	MDP_AOP16_QW1
W	BTS	STRBUF,NULL	SIMD 8	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_QW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD 16	1	MSD1R_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_QW2	MDP_AOP16_QW1
W	BTS	STRBUF,NULL	SIMD 16	1	MSD1W_DWAI3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_QW2	{Forbidden}

Programming Note	
Context:	QWord Scaled 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.	

OWord 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
OW	OW	R/W	A64	BUFFER	4x2	1	(Base+U{Slot})→OW[0]	Canonical	SM4x2
OW	OW	R/W	A64	BUFFER	8	1	(Base+U{Slot})→QW[0]	Canonical	SM

This message performs OWord atomic integer CMPWR on untyped surfaces using 8 or 16 offsets. Only A64 stateless addresses are supported. The surface format is RAW and the surface type is SURFTYPE_BUFFER. One QWord is accessed beginning at the byte address determined. Each U offset must be OWord-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	SIMD 4x2	1	MSD1R_A64_QWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_OW2	MDP_A64_AOP4x2_OW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_QWAI3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_A64_AOP4x2_OW2	{Forbidden}
R	A64	BUFFER	SIMD 8	1	MSD1R_A64_QWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_OW2	MDP_OW8
W	A64	BUFFER	SIMD 8	1	MSD1W_A64_QWAI3	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_OW2	{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	4x2	1	(Base+U{Slot})→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})→DW[0]	Surface	SMPSM
DW	DW	R/W	SLM	BUFFER	4x2	1	(Base+U{Slot})→DW[0]	Wrap	SM4x2
DW	DW	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})→DW[0]	Wrap	SMPSM
DW	DW	R/W	A32	BUFFER	4x2	1	(Base+Buffer+U{Slot})→DW[0]	GenState	SM4x2
DW	DW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})→DW[0]	GenState	SMPSM
DW	DW	R/W	A64	BUFFER	4x2	1	(0+U{Slot})→DW[0]	Canonical	SM4x2
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. 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	SIMD4x2	1	MSD1R_DWAF2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	BUFFER,NULL	SIMD4x2	1	MSD1W_DWAF2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAF2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAF2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAF2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAF2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	BUFFER,NULL	SIMD4x2	1	MSD1R_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	BTS	BUFFER,NULL	SIMD4x2	1	MSD1W_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	SLM	BUFFER	SIMD 4x2	1	MSD1R_DWAF2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	SLM	BUFFER	SIMD 4x2	1	MSD1W_DWAF2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	SLM	BUFFER	SIMD 8	1	MSD1R_DWAF2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD 8	1	MSD1W_DWAF2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD 16	1	MSD1R_DWAF2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD 16	1	MSD1W_DWAF2	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD 4x2	1	MSD1R_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	SLM	BUFFER	SIMD 4x2	1	MSD1W_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	SLM	BUFFER	SIMD 8	1	MSD1R_DWAF3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD 8	1	MSD1W_DWAF3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	SLM	BUFFER	SIMD 16	1	MSD1R_DWAF3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD 16	1	MSD1W_DWAF3	{Opt} MH1_SLM_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A32	BUFFER	SIMD 4x2	1	MSD1R_DWAF2_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	A32	BUFFER	SIMD 4x2	1	MSD1W_DWAF2_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{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	SIMD 8	1	MSD1R_DWAF2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD 8	1	MSD1W_DWAF2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD 16	1	MSD1R_DWAF2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD 16	1	MSD1W_DWAF2	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD 4x2	1	MSD1R_DWAF3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	A32	BUFFER	SIMD 4x2	1	MSD1W_DWAF3_4x2	{Opt} MH1_A32	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	A32	BUFFER	SIMD 8	1	MSD1R_DWAF3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD 8	1	MSD1W_DWAF3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD 16	1	MSD1R_DWAF3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD 16	1	MSD1W_DWAF3	{Opt} MH1_A32_PSM	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD1R_A64_DWAF2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_DWAF2_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	A64	BUFFER	SIMD 8	1	MSD1R_A64_DWAF2	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A64	BUFFER	SIMD 8	1	MSD1W_A64_DWAF2	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD 4x2	1	MSD1R_A64_DWAF3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	A64	BUFFER	SIMD 4x2	1	MSD1W_A64_DWAF3_4x2	{Forbidden}	MAP64b_USU_SIMD4x2	MDP_AOP4x2_DW2	{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	SIMD8	1	MSD1R_A64_DWAF3	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAF3	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}

DWord Scaled 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	STRBUF,NULL	4x2	1	(Base+(U{Slot}*Pitch)+V{Slot})→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	STRBUF,NULL	8, 16	1	(Base+(U{Slot}*Pitch)+V{Slot})→DW[0]	Surface	SMPSM

This message performs atomic float operations on untyped surfaces using 8 or 16 offsets. One DWord is accessed beginning at the byte address determined. Each U offset is scaled by Pitch. Both Surface Pitch and V 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	STRBUF,NULL	SIMD4x2	1	MSD1R_DWAF2_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	STRBUF,NULL	SIMD4x2	1	MSD1W_DWAF2_4x2	{Opt}MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	STRBUF,NULL	SIMD8	1	MSD1R_DWAF2	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD8	1	MSD1W_DWAF2	{Opt}MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	STRBUF,NULL	SIMD16	1	MSD1R_DWAF2	{Opt}MH1_BTS_PSM	MAP32b_USUV_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
W	BTS	STRBUF,NULL	SIMD16	1	MSD1W_DWAF2	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	STRBUF,NULL	SIMD4x2	1	MSD1R_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	BTS	STRBUF,NULL	SIMD4x2	1	MSD1W_DWAF3_4x2	{Opt} MH_IGNORE	MAP32b_USUV_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD8	1	MSD1R_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	STRBUF,NULL	SIMD8	1	MSD1W_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	STRBUF,NULL	SIMD16	1	MSD1R_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	STRBUF,NULL	SIMD16	1	MSD1W_DWAF3	{Opt} MH1_BTS_PSM	MAP32b_USUV_SIMD16	MDP_AOP16_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	4x2	1	(Surface[U, V, R, LOD])→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	BUFFER, NULL	4x2	1	(Surface[U])→DW[0]	Surface	SM4x2
DW	DW	R/W	BTS	1D, 2D, 3D, CUBE	8	1	(Surface[U, V, R, LOD])→DW[0]	Surface	SGPSM
DW	DW	R/W	BTS	BUFFER, NULL	8	1	(Surface[U])→DW[0]	Surface	SGPSM

This message performs atomic integer operations on typed surfaces using 8 offsets. The surface format must be R32_UINT or R32_SINT, 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	SIMD 4x2	1	MSD1R_DWTAI1_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	BTS	1D, 2D, 3D, CUBE	SIMD 4x2	1	MSD1W_DWTAI1_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	{Forbidden}	{Forbidden}
R	BTS	BUFFER, NULL	SIMD 4x2	1	MSD1R_DWTAI1_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	MDP_AOP4x2_DW1
W	BTS	BUFFER, NULL	SIMD 4x2	1	MSD1W_DWTAI1_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	{Forbidden}	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD 8	1	MSD1R_DWTAI1	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	1D, 2D, 3D, CUBE	SIMD 8	1	MSD1W_DWTAI1	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER, NULL	SIMD 8	1	MSD1R_DWTAI1	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD 8	1	MSD1W_DWTAI1	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD 4x2	1	MSD1R_DWTAI2_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	1D, 2D, 3D, CUBE	SIMD 4x2	1	MSD1W_DWTAI2_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	BUFFER, NULL	SIMD 4x2	1	MSD1R_DWTAI2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	BUFFER, NULL	SIMD 4x2	1	MSD1W_DWTAI2_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	1D, 2D, 3D,	SIMD 8	1	MSD1R_DWTAI2	{Opt} MH1_BTS_	MAP32b_TS_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
		CUBE				PSM			
W	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1W_DWTAI2	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER, NULL	SIMD8	1	MSD1R_DWTAI2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	1	MSD1W_DWTAI2	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD4x2	1	MSD1R_DWTAI3_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	BTS	1D, 2D, 3D, CUBE	SIMD4x2	1	MSD1W_DWTAI3_4x2	{Opt} MH_IGNORE	MAP32b_TS_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	BUFFER, NULL	SIMD4x2	1	MSD1R_DWTAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	MDP_AOP4x2_DW1
W	BTS	BUFFER, NULL	SIMD4x2	1	MSD1W_DWTAI3_4x2	{Opt} MH_IGNORE	MAP32b_USU_SIMD4x2	MDP_AOP4x2_DW2	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1R_DWTAI3	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1W_DWTAI3	{Opt} MH1_BTS_PSM	MAP32b_TS_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER, NULL	SIMD8	1	MSD1R_DWTAI3	{Opt} MH1_BTS_PSM	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	1	MSD1W_DWTAI3	{Opt} MH1_BTS_PSM	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.	

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	4x2	1	(Surface Append Buffer)→DW[0]	Ignored	SM4x2
DW	DW	R/W	BTS	Any	8	1	(Surface Append Buffer)→DW[0]	Ignored	SGPSM, 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	SIMD 4x2	1	MSD1R_DWAC1_4x2	MH_IGNORE	{Forbidden}	{Forbidden}	MDP_AOP4x2_DW1
W	BTS	Any	SIMD 4x2	1	MSD1W_DWAC1_4x2	MH_IGNORE	{Forbidden}	{Forbidden}	{Forbidden}
R	BTS	Any	SIMD 8	1	MSD1R_DWAC1	MH1_BTS_PSM	{Forbidden}	{Forbidden}	MDP_DW_SIMD8
W	BTS	Any	SIMD 8	1	MSD1W_DWAC1	MH1_BTS_PSM	{Forbidden}	{Forbidden}	{Forbidden}
R	BTS	Any	SIMD 4x2	1	MSD1R_DWAC2_4x2	{Forbidden}	{Forbidden}	MDP_AOP4x2_DW1	MDP_AOP4x2_DW1
W	BTS	Any	SIMD 4x2	1	MSD1W_DWAC2_4x2	{Forbidden}	{Forbidden}	MDP_AOP4x2_DW1	{Forbidden}
R	BTS	Any	SIMD 8	1	MSD1R_DWAC2	{Forbidden}	{Forbidden}	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	Any	SIMD 8	1	MSD1W_DWAC2	{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.

Transpose 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	2D, NULL	Height	Width	(Surface[X, Y])→DW[0]	Media	Ignored

This message enables a rectangular block of DWords to be read from the source surface and written in transposed order into the GRF. 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 reads return a replicated boundary pixel. More details on bounds checking is described in Addressing 2D Media Surfaces in this volume.

Applications:

Transpose a 2D media block of 32bpe pixels.

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, 2, 4, 8 Rows	{1, 2, 4, 8} Width	MSD1R_TT	MH1_T	{Forbidden}	{Forbidden}	{1, 2, 4, 8} MDP_DW_SIMD8

The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Accesses are allowed to SURFTYPE_NULL and will return 0. More details on 2D surfaces used by this message is described in Addressing 2D Media Surfaces in this volume

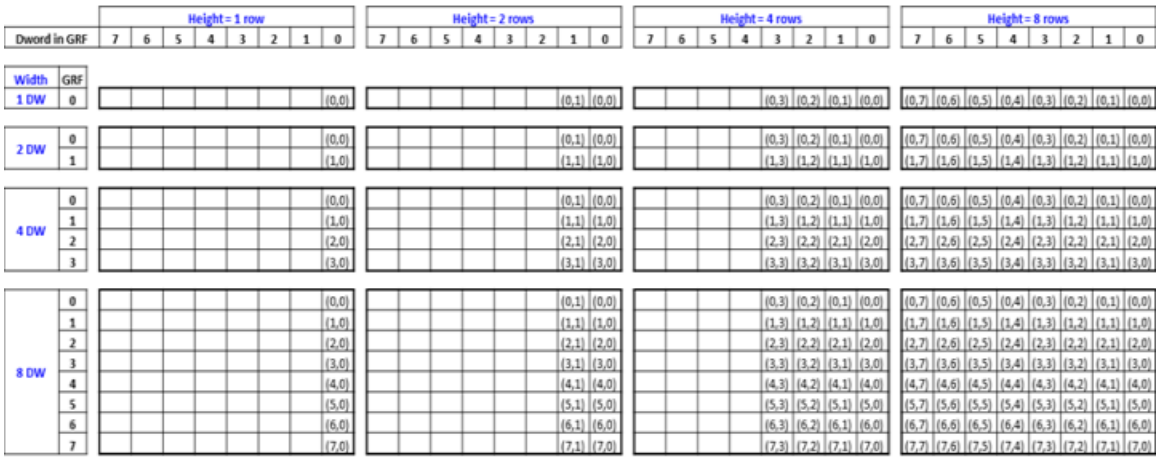
Vertical stride, offset, and boundary clamping media surface state parameters are supported. The surface format must be 32bpe (Dword size) as described in 2D Media Surface Formats. The raw data from the surface is returned to the thread without any format conversion operation.

Block Width and Block Height

The Block Width (in Dwords) and Block Height (in rows) are specified in the message header.

The layout of the data payload depends on the Block Height and Block Width. Each row's first Dword is aligned to the corresponding Dword of the first register, and every next Dword in the X dimension of the block is aligned to the corresponding Dword of the next register. The figure below illustrates the layout of the data returned in the register in (x,y) offsets from the (X,Y) start of the block. Dwords not used in a GRF are not modified.

Layout of Transposed Rows in Registers



Programming Note	
Context:	Transpose Read Message

- Tile X is not supported: the supported tiling modes are Linear and Tile Y/YF/YS.
- The surface base address must be 32-byte aligned.
- The surface width must be a multiple of DWords.
- Pitch must be a multiple of 64 bytes when the surface is linear.
- The **X Offset** must be a multiple of the block width in bytes. The **Y Offset** must be a multiple of the block height.
- Transpose Read is not supported on compressible media surfaces.

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}	{1-8} MDP_HW_B
W	BTS	2D, NULL	1 block	Height rows x Width bytes	MSD1W_MB	MH_MB	{Forbidden}	{1-8} MDP_HW_B	{Forbidden}
R	BTS	2D, NULL	1 block	Height rows x Width bytes	MSD_SC_MB	MH_MB	{Forbidden}	{Forbidden}	{1-8} MDP_HW_B

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.

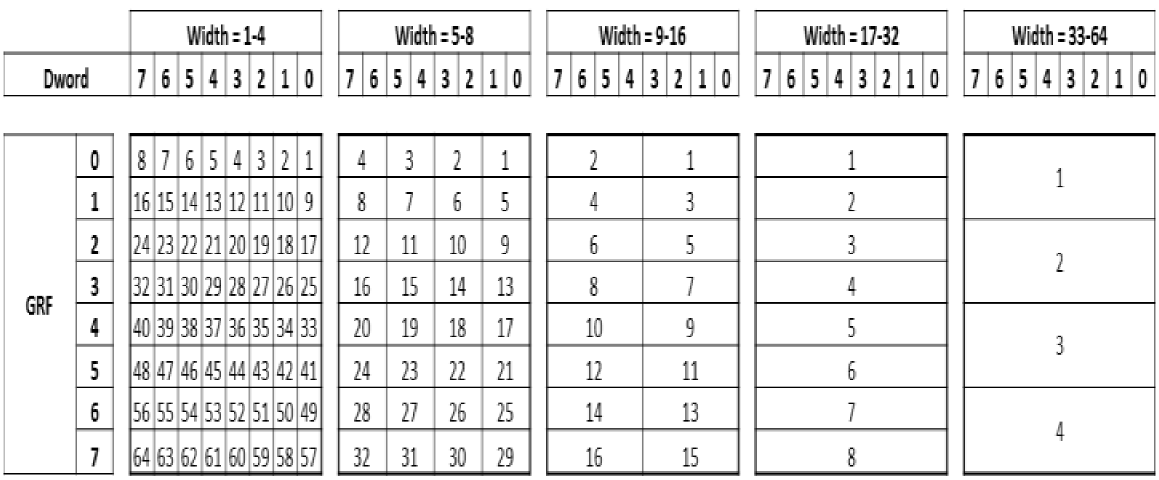
Block Width and Block Height

The maximum Block Height is limited by the Block Width. The maximum data size supported in this message is 64 Dwords (256 Bytes). Blocks wider than 32 Bytes are supported only with write operations 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



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.

Programming Note	
Context:	Media Block Read/Write Messages

Restrictions Media Block Write:

- For media block writes, both X Offset and Block Width must be DWord-aligned.
- Media block writes to Linear or TileX surfaces must have a height of 16 or less.
- Media block writes are not supported on compressible media surfaces.

Pixel 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 a fixed block size of 4 rows by 8 Dwords, and a Pixel Mask to enable writing of the individual Dwords.

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 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
W	BTS	2D, NULL	1 block	4 rows x 8 DW	MSD1W_MB	MH_MBPM	{Forbidden}	{4} MDP_HW_B	{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.

All tile modes are supported: 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, with 1 row per register. The Pixel Mask applies to the 2x2 square tiles (UL, UR, LL, LR), which themselves tiled (UL, UR, LL, LR) and then repeated on the right for the remaining 16-bits to cover a 4 row 8 column area. When the Pixel Mask bit is set, the Dword is written. The figure below illustrates the layout of the data and corresponding mask bits.

Data and Pixel Mask Layout

Dword	7	6	5	4	3	2	1	0

GRF Row Data									
	0	7	6	5	4	3	2	1	0
	1	15	14	13	12	11	10	9	8
	2	23	22	21	20	19	18	17	16
	3	31	30	29	28	27	26	25	24

Corresponding Pixel Mask Bits								
21	20	17	16	5	4	1	0
23	22	19	18	7	6	3	2
29	28	25	24	13	12	9	8
31	30	27	26	15	14	11	10

Programming Note	
Context:	Pixel Masked Media Block Write Message

- X Offset must be DWord-aligned.
- Pitch must be a multiple of 64 bytes when the surface is linear.
- The surface base address must be 32-byte aligned.
- 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}	{1-8} MDP_HW_B	{Forbidden}

The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Accesses are allowed to SURFTYPE_NULL, writes will be ignored. More details on 2D surfaces used by this message is described in Addressing 2D Media Surfaces in this volume.

See 2D Media Surface Formats in this volume for limitations on surface formats supported by this message.

Block Width and Block Height

The maximum Block Height is limited by the Block Width. The maximum data size supported in this message is 64 Dwords (256 Bytes).

Block Width (bytes)	Block Height (rows)	Tile Modes Supported
1-4	1-64	Linear, TileX, TileY/YF/YS
5-8	1-32	Linear, TileX, TileY/YF/YS
9-16	1-16	Linear, TileX, TileY/YF/YS
17-32	1-8	Linear, TileX, TileY/YF/YS

The data payload is aligned with the upper left pixel going into the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the Block Width. The Byte Mask applies horizontally to each row of output. When the Byte Mask bit is set, the byte is written. The figure below illustrates the layout of the row data and corresponding mask bits.

Row and Byte Mask Layout

Dword		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32							
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
GRF	0	8	7	6	5	4	3	2	1	4	3	2	1					2	1							1							
	1	16	15	14	13	12	11	10	9	8	7	6	5					4	3							2							
	2	24	23	22	21	20	19	18	17	12	11	10	9					6	5							3							
	3	32	31	30	29	28	27	26	25	16	15	14	13					8	7							4							
	4	40	39	38	37	36	35	34	33	20	19	18	17					10	9							5							
	5	48	47	46	45	44	43	42	41	24	23	22	21					12	11							6							
	6	56	55	54	53	52	51	50	49	28	27	26	25					14	13							7							
	7	64	63	62	61	60	59	58	57	32	31	30	29					16	15							8							
Byte Mask		Bits 3:0 for each Row								Bits 7:0 for each Row								Bits 15:0 for each Row								Bits 31:0 for each Row							

Programming Note	
Context:	Byte Masked Media Block Write Message

- Linear or TileX surfaces must have a block height <= 16.
- Both X Offset and Block Width must be DWord-aligned.
- The surface base address must be 32-byte aligned.
- Pitch must be a multiple of 64 bytes when the surface is linear.
- Media block writes are not supported on compressible media surfaces.

Merged 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	BTS	2D, NULL	1 block	Width x Height	(Surface[X, Y])→DW[0]	Media	Ignored

This message enables a rectangular block of data samples to be read from the source surface, and written into the GRF in a spread-out form without modifying any other portions of the GRF. The message specifies as inputs a signed (X, Y) coordinate into the 2D surface, a block size (Width, Height), a Register Pitch Control, and a Sub-Register Offset.

The execution mask is ignored. Out-of-bounds reads return a replicated boundary pixel. Media bounds checking is described in Addressing 2D Media Surfaces in this volume.

Applications:

Allow software to assemble multiple media block reads directly into a shared GRF register set. Multiple media block read messages sharing the same Register Pitch Control but with different Sub-Register Offset can fill in the same set of GRF registers with interleaved rows of media blocks.

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_MBM	{Forbidden}	{Forbidden}	{1-31} MDP_HW_B

The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Reads are allowed to SURFTYPE_NULL and will return 0. More details on 2D surfaces used by this message are 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 layout of the read data payload depends on the Block Height, Block Width, Register Pitch, and Sub-Register Offset. The upper left pixel is aligned to the least significant bits of the first register, and the register pitch is equal to the next multiple of the Register Pitch Control and Data Width. Finally, the data is aligned to the Sub-Register Offset of the first register.

The maximum Block Height is limited by the Block Width. The maximum packed data size supported in this message is 64 Dwords (256 Bytes).When Register Pitch Control is 4 rows and the Block Width <= 8, the maximum Block Height is additionally limited by the maximum allowed message response of 31 GRF registers.

Block Width (bytes)	Register Pitch Control	Sub-Register Offset	Block Height (rows) Supported	Tile Modes Supported
1-4	0	0,4,8,12,16,20,24,28	1	Linear, TileX, TileY/YF/YS
	2	0,4	1-64	Linear, TileX, TileY/YF/YS
	2	16,20	1-2	Linear, TileX, TileY/YF/YS
	4	0,4,8,12	1-62	Linear, TileX, TileY/YF/YS
5-8	0	0,8,16,24	1	Linear, TileX, TileY/YF/YS
	2	0,8	1-32	Linear, TileX, TileY/YF/YS

Block Width (bytes)	Register Pitch Control	Sub-Register Offset	Block Height (rows) Supported	Tile Modes Supported
	4	0,8	1-31	Linear, TileX, TileY/YF/YS
9-16	0	0,16	1	Linear, TileX, TileY/YF/YS
	2	16	1-16	Linear, TileX, TileY/YF/YS
	4	16	1-16	Linear, TileX, TileY/YF/YS
17-32	2	0	1-8	Linear, TileX, TileY/YF/YS
	4	0	1-8	Linear, TileX, TileY/YF/YS

Certain combinations of Register Pitch and Sub-Register Offset allow interleaving multiple rows into one block. When only 1 GRF is being updated, then additional Sub-Register Offset combinations are supported. The figures below illustrate for each of the block width cases, the layout of the rows in the registers and which combinations of Register Pitch Control and Sub-Register Offset are supported.

Two-way Row Interleave into Registers

Register Pitch Control	2	Sub-Register Offset (SRO) selections			
		SRO = 0	SRO = 0	SRO = 0	SRO = 0
		SRO = 4	SRO = 8	SRO = 16	GRF+1, SRO = 0

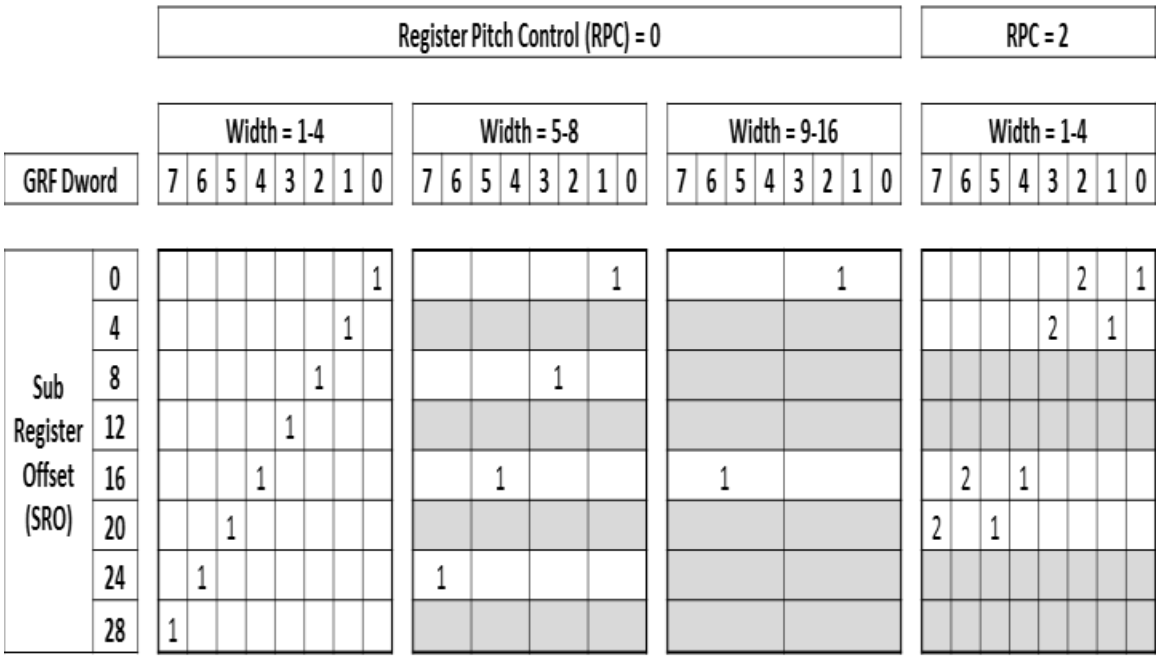
		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32							
Dword		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
GRF	0	4	4	3	3	2	2	1	1	2	2	1	1					1	1							1							
	1	8	8	7	7	6	6	5	5	4	4	3	3					2	2							1							
	2	12	12	11	11	10	10	9	9	6	6	5	5					3	3							2							
	3	16	16	15	15	14	14	13	13	8	8	7	7					4	4							2							
	4	20	20	19	19	18	18	17	17	10	10	9	9					5	5							3							
	5	24	24	23	23	22	22	21	21	12	12	11	11					6	6							3							
	6	28	28	27	27	26	26	25	25	14	14	13	13					7	7							4							
	7	32	32	31	31	30	30	29	29	16	16	15	15					8	8							4							
	8	36	36	35	35	34	34	33	33	18	18	17	17					9	9							5							
	9	40	40	39	39	38	38	37	37	20	20	19	19					10	10							5							
	10	44	44	43	43	42	42	41	41	22	22	21	21					11	11							6							
	11	48	48	47	47	46	46	45	45	24	24	23	23					12	12							6							
	12	52	52	51	51	50	50	49	49	26	26	25	25					13	13							7							
	13	56	56	55	55	54	54	53	53	28	28	27	27					14	14							7							
	14	60	60	59	59	58	58	57	57	30	30	29	29					15	15							8							
	15	64	64	63	63	62	62	61	61	32	32	31	31					16	16							8							

Four-way Row Interleave into Registers

Register Pitch Control	4	Sub-Register Offset (SRO) selections			
		SRO = 0	SRO = 0	SRO = 0	SRO = 0
		SRO = 4	SRO = 8	SRO = 16	GRF+1, SRO = 0
		SRO = 8	SRO = 16	GRF+1, SRO = 0	GRF+2, SRO = 0
		SRO = 12	SRO = 24	GRF+1, SRO = 16	GRF+3, SRO = 0

Dword		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32							
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
GRF	0	2	2	2	2	1	1	1	1	1	1	1	1					1	1							1							
	1	4	4	4	4	3	3	3	3	2	2	2	2					1	1							1							
	2	6	6	6	6	5	5	5	5	3	3	3	3					2	2							1							
	3	8	8	8	8	7	7	7	7	4	4	4	4					2	2							1							
	4	10	10	10	10	9	9	9	9	5	5	5	5					3	3							2							
	5	12	12	12	12	11	11	11	11	6	6	6	6					3	3							2							
	6	14	14	14	14	13	13	13	13	7	7	7	7					4	4							2							
	7	16	16	16	16	15	15	15	15	8	8	8	8					4	4							2							
	8	18	18	18	18	17	17	17	17	9	9	9	9					5	5							3							
	9	20	20	20	20	19	19	19	19	10	10	10	10					5	5							3							
	10	22	22	22	22	21	21	21	21	11	11	11	11					6	6							3							
	11	24	24	24	24	23	23	23	23	12	12	12	12					6	6							3							
	12	26	26	26	26	25	25	25	25	13	13	13	13					7	7							4							
	13	28	28	28	28	27	27	27	27	14	14	14	14					7	7							4							
	14	30	30	30	30	29	29	29	29	15	15	15	15					8	8							4							
	15	32	32	32	32	31	31	31	31	16	16	16	16					8	8							4							
	16	34	34	34	34	33	33	33	33	17	17	17	17					9	9							5							
	17	36	36	36	36	35	35	35	35	18	18	18	18					9	9							5							
	18	38	38	38	38	37	37	37	37	19	19	19	19					10	10							5							
	19	40	40	40	40	39	39	39	39	20	20	20	20					10	10							5							
	20	42	42	42	42	41	41	41	41	21	21	21	21					11	11							6							
	21	44	44	44	44	43	43	43	43	22	22	22	22					11	11							6							
	22	46	46	46	46	45	45	45	45	23	23	23	23					12	12							6							
	23	48	48	48	48	47	47	47	47	24	24	24	24					12	12							6							
	24	50	50	50	50	49	49	49	49	25	25	25	25					13	13							7							
	25	52	52	52	52	51	51	51	51	26	26	26	26					13	13							7							
	26	54	54	54	54	53	53	53	53	28	28	28	28					14	14							7							
	28	56	56	56	56	55	55	55	55	28	28	28	28					14	14							7							
	28	58	58	58	58	57	57	57	57	29	29	29	29					15	15							8							
	29	60	60	60	60	59	59	59	59	30	30	30	30					15	15							8							
	30	62	62	62	62	61	61	61	61	31	31	31	31					16	16							8							
	31																	16	16							8							

Merging Rows into 1 Register



Programming Note	
Context:	Merged Media Block Read/Write Messages

- 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 should be aligned to the size of pixels stored in the surface. For a surface with 8bpp pixels for example, the block width and offset can be byte-aligned. For a surface with 16bpp pixels, it is word-aligned.

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	N/A	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	N/A	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.

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 Gen caches. These controls are generally not used.

Programming Note	
Context:	Memory Fence Message
• 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.	

Workaround-Memory Fence Message Commit Enable

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

Data Port 2 Message Descriptors
Data Port 2 messages do not have a BTI field, and encode whether the access is to SLM or A32 memory based on the settings of the Message Header Present and SBSO (Sideband Scaled Offset) fields. The SBSO field is encoded in the SENDS instruction's extended descriptor on bits [31:16].

Data Port 2 Extended Message Descriptor

Data Port 0 Message Specific Descriptors

This section contains the message specific descriptors for Data Port 0.

MT_DP0 - Data Port 0 Message Types

Scattered Read/Write Messages

Message Descriptor
MSD0R_BS - Byte Scattered Read MSD
MSD0W_BS - Byte Scattered Write MSD
MSD0R_DWS - Dword Scattered Read MSD
MSD0W_DWS - Dword Scattered Write MSD

Block Read/Write Messages

Message Descriptor
MSD0R_OWB - Oword Block Read MSD
MSD0W_OWB - Oword Block Write MSD
MSD0R_OWAB - Oword Aligned Block Read MSD
MSD0R_OWDB - Oword Dual Block Read MSD
MSD0W_OWDB - Oword Dual Block Write MSD
MSD0R_HWB - Scratch Block Read MSD
MSD0W_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_OWDB - A64 Oword Dual Block Read MSD
MSD1W_A64_OWDB - A64 Oword Dual Block Write 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_TT - Media Transpose Read MSD
MSD1R_MB - Media Block Read MSD
MSD1W_MB - Media Block Write MSD

A32 Dword Untyped Atomic Operations

Descriptor
MSD1R_DWAI1 - Dword Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_DWAI2 - Dword Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_DWAI3 - Dword Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_DWAI1 - Dword Untyped Atomic Integer Unary Write Only Operation MSD
MSD1W_DWAI2 - Dword Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_DWAI3 - Dword Untyped Atomic Integer Trinary Write Only Operation MSD
MSD1R_DWAI1_4x2 - Dword SIMD4x2 Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_DWAI2_4x2 - Dword SIMD4x2 Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_DWAI3_4x2 - Dword SIMD4x2 Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_DWAI1_4x2 - Dword SIMD4x2 Untyped Atomic Integer Unary Write Only Operation MSD

Descriptor
MSD1W_DWAI2_4x2 - Dword SIMD4x2 Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_DWAI3_4x2 - Dword SIMD4x2 Untyped Atomic Integer Trinary Write Only Operation MSD
MSD1R_DWAF2 - Dword Untyped Atomic Float Binary with Return Data Operation MSD
MSD1R_DWAF3 - Dword Untyped Atomic Float Trinary with Return Data Operation MSD
MSD1W_DWAF2 - Dword Untyped Atomic Float Binary Write Only Operation MSD
MSD1W_DWAF3 - Dword Untyped Atomic Float Trinary Write Only Operation MSD
MSD1R_DWAF2_4x2 - Dword SIMD4x2 Untyped Atomic Float Binary with Return Data Operation MSD
MSD1R_DWAF3_4x2 - Dword SIMD4x2 Untyped Atomic Float Trinary with Return Data Operation MSD
MSD1W_DWAF2_4x2 - Dword SIMD4x2 Untyped Atomic Float Binary Write Only Operation MSD
MSD1W_DWAF3_4x2 - Dword SIMD4x2 Untyped Atomic Float Trinary Write Only Operation MSD

A64 Dword Untyped Atomic Operations

Descriptor
MSD1R_A64_DWAI1 - A64 Dword Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_A64_DWAI2 - A64 Dword Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_A64_DWAI3 - A64 Dword Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_A64_DWAI1 - A64 Dword Untyped Atomic Integer Unary Write Only Operation MSD
MSD1W_A64_DWAI2 - A64 Dword Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_A64_DWAI3 - A64 Dword Untyped Atomic Integer Trinary Write Only Operation MSD
MSD1R_A64_DWAI1_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_A64_DWAI2_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_A64_DWAI3_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_A64_DWAI1_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Unary Write Only Operation MSD
MSD1W_A64_DWAI2_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_A64_DWAI3_4x2 - A64 Dword SIMD4x2 Untyped Atomic Integer Trinary Write Only Operation MSD
MSD1R_A64_DWAF2 - A64 Dword Untyped Atomic Float Binary with Return Data Operation MSD
MSD1R_A64_DWAF3 - A64 Dword Untyped Atomic Float Trinary with Return Data Operation MSD
MSD1W_A64_DWAF2 - A64 Dword Untyped Atomic Float Binary Write Only Operation MSD
MSD1W_A64_DWAF3 - A64 Dword Untyped Atomic Float Trinary Write Only Operation MSD
MSD1R_A64_DWAF2_4x2 - A64 Dword SIMD4x2 Untyped Atomic Float Binary with Return Data Operation MSD
MSD1R_A64_DWAF3_4x2 - A64 Dword SIMD4x2 Untyped Atomic Float Trinary with Return Data Operation MSD
MSD1W_A64_DWAF2_4x2 - A64 Dword SIMD4x2 Untyped Atomic Float Binary Write Only Operation MSD
MSD1W_A64_DWAF3_4x2 - A64 Dword SIMD4x2 Untyped Atomic Float Trinary Write Only Operation MSD

A64 Qword Untyped Atomic Operations

Descriptor
MSD1R_A64_QWAI1 - A64 Qword Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_A64_QWAI2 - A64 Qword Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_A64_QWAI3 - A64 Qword Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_A64_QWAI1 - A64 Qword Untyped Atomic Integer Unary Write Only Operation MSD
MSD1W_A64_QWAI2 - A64 Qword Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_A64_QWAI3 - A64 Qword Untyped Atomic Integer Trinary Write Only Operation MSD
MSD1R_A64_QWAI1_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Unary with Return Data Operation MSD
MSD1R_A64_QWAI2_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Binary with Return Data Operation MSD
MSD1R_A64_QWAI3_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Trinary with Return Data Operation MSD
MSD1W_A64_QWAI1_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Unary Write Only Operation MSD
MSD1W_A64_QWAI2_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Binary Write Only Operation MSD
MSD1W_A64_QWAI3_4x2 - A64 Qword SIMD4x2 Untyped Atomic Integer Trinary Write Only Operation MSD

Dword Typed Atomic Operations

Descriptor
MSD1R_DWTAI1 - Dword Typed Atomic Integer Unary with Return Data Operation MSD
MSD1R_DWTAI2 - Dword Typed Atomic Integer Binary with Return Data Operation MSD
MSD1R_DWTAI3 - Dword Typed Atomic Integer Trinary with Return Data Operation MSD
MSD1W_DWTAI1 - Dword Typed Atomic Integer Unary Write Only Operation MSD
MSD1W_DWTAI2 - Dword Typed Atomic Integer Binary Write Only Operation MSD
MSD1W_DWTAI3 - Dword Typed Atomic Integer Trinary Write Only Operation MSD
MSD1R_DWTAI1_4x2 - Dword SIMD4x2 Typed Atomic Integer Unary with Return Data Operation MSD
MSD1R_DWTAI2_4x2 - Dword SIMD4x2 Typed Atomic Integer Binary with Return Data Operation MSD
MSD1R_DWTAI3_4x2 - Dword SIMD4x2 Typed Atomic Integer Trinary with Return Data Operation MSD
MSD1W_DWTAI1_4x2 - Dword SIMD4x2 Typed Atomic Integer Unary Write Only Operation MSD
MSD1W_DWTAI2_4x2 - Dword SIMD4x2 Typed Atomic Integer Binary Write Only Operation MSD
MSD1W_DWTAI3_4x2 - Dword SIMD4x2 Typed Atomic Integer Trinary Write Only Operation MSD

Dword Atomic Counter Operations

Descriptor
MSD1R_DWAC1 - Dword Atomic Counter Unary with Return Data Operation MSD
MSD1R_DWAC2 - Dword Atomic Counter Binary with Return Data Operation MSD
MSD1W_DWAC1 - Dword Atomic Counter Unary Write Only Operation MSD
MSD1W_DWAC2 - Dword Atomic Counter Binary Write Only Operation MSD

Descriptor
MSD1R_DWAC1_4x2 - Dword SIMD4x2 Atomic Counter Unary with Return Data Operation MSD
MSD1R_DWAC2_4x2 - Dword SIMD4x2 Atomic Counter Binary with Return Data Operation MSD
MSD1W_DWAC1_4x2 - Dword SIMD4x2 Atomic Counter Unary Write Only Operation MSD
MSD1W_DWAC2_4x2 - Dword SIMD4x2 Atomic Counter Binary Write Only Operation MSD

Data Port 2 Message Specific Descriptors

This section contains the message specific descriptors for Data Port 2.

MT_DP2 - Data Port 2 Message Types

A32 and SLM Scaled Read/Write

Message Descriptor
MSD2R_BS - Byte Scaled Read MSD
MSD2W_BS - Byte Scaled Write MSD
MSD2R_US - Scaled Untyped Surface Read MSD
MSD2W_US - Scaled Untyped Surface Write MSD

A64 Scaled Read/Write

Message Descriptor
MSD2R_A64_BS - A64 Byte Scaled Read MSD
MSD2W_A64_BS - A64 Byte Scaled Write MSD
MSD2R_A64_DWS - A64 Dword Scaled Read MSD
MSD2W_A64_DWS - A64 Dword Scaled Write MSD
MSD2R_A64_QWS - A64 Qword Scaled Read MSD
MSD2W_A64_QWS - A64 Qword Scaled Write MSD
MSD2R_A64_US - A64 Scaled Untyped Surface Read MSD
MSD2W_A64_US - A64 Scaled Untyped Surface 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/Write

Message Descriptor
MSD_CC_DWS - Constant Cache Dword Scattered Read MSD
MSD_CC_OWB - Constant Cache Oword Block Read MSD
MSD_CC_OWUB - Constant Cache Oword Unaligned Block Read MSD

Message Descriptor
MSD_CC_OWDB - Constant Cache Oword Dual 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

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:

Pixel Sample Mask	This specifies the channel mask for SIMD8 and SIMD16 operations. The default value is 0FFFFh when the message header is not present.
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.

Scaled untyped surface and byte-aligned operations are provided on Data Port 2 for the Read/Write data cache. This Message Header is programmed to provide:

Pixel Sample Mask	This specifies the channel mask for SIMD8 and SIMD16 operations for surface operations.
Surface Pitch	Scale factor for stateless messages.
Buffer Base Address	A base address used for stateless address models.

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_OWDB - A64 Dual Oword Block Message Header
MH_A64_HWB - A64 Hword Block Message Header
MH1_BTS_PSM - Surface Pixel Mask Message Header
MH1_SLM_PSM - SLM Surface Pixel Mask Message Header
MH1_A32_PSM - Stateless Surface Pixel Mask Message Header
MH1_A32 - Stateless Surface Message Header
MH_T - Transpose Message Header
MH_MB - Normal Media Block Message Header
MH_MBPM - Pixel Masked Media Block Message Header
MH_MBBM - Byte Masked Media Block Message Header
MH_MBM - Merged Media Block Message Header

Data Port 2 Message Headers

This section contains the message headers for Data Port 2.

Message Header
MH2_A32 - Stateless Scaled Data Port 2 Message Header
MH2_A32_PSM - Stateless Scaled Data Port 2 Pixel Mask Message Header
MH2_A64 - A64 Scaled Data Port 2 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_SIMD4x2 - SIMD4x2 32-Bit Address Payload
MAP32b_SIMD8 - SIMD8 32-Bit Address Payload
MAP32b_SIMD16 - SIMD16 32-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_SIMD4x2 - SIMD4x2 Untyped BUFFER Surface 32-Bit Address Payload
MAP32b_USU_SIMD8 - SIMD8 Untyped BUFFER Surface 32-Bit Address Payload
MAP32b_USU_SIMD16 - SIMD16 Untyped BUFFER Surface 32-Bit Address Payload
MAP32b_USUV_SIMD4x2 - SIMD4x2 Untyped STRBUF Surface 32-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

64 Bit Untyped Surface Address Payloads

Address Payload
MAP64b_USU_SIMD4x2 - SIMD4x2 Untyped BUFFER Surface 64-Bit 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_SIMD4x2 - SIMD4x2 Typed Surface 32-Bit Address Payload
MAP32b_TS_SIMD8 - SIMD8 Typed Surface 32-Bit Address Payload
MAP32b_MSAA_TS_SIMD4x2 - SIMD4x2 MSAA Typed Surface 32-Bit Address Payload
MAP32b_MSAA_TS_SIMD8 - SIMD8 MSAA Typed Surface 32-Bit Address Payload

Address Payload
MAP32b_RSI - Read Surface Info 32-Bit 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
MDP_OWD1 - Oword 1 Dual Block Data Payload
MDP_OWD4 - Oword 4 Dual 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

Dword SIMD Data Payloads

Payload
MDP_DW_SIMD8 - Dword SIMD8 Data Payload
MDP_DW_SIMD16 - Dword SIMD16 Data Payload
MDP_DW_SIMD4x2 - Dword SIMD4x2 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_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
MDP_AOP4x2_DW1 - Dword SIMD4x2 Atomic Operation Message Data Payload
MDP_AOP4x2_DW2 - Dword SIMD4x2 Atomic CMPWR Message Data Payload
MDP_AOP4x2_QW1 - Qword SIMD4x2 Atomic Operation Message Data Payload
MDP_AOP4x2_QW2 - Qword SIMD4x2 Atomic CMPWR8B Message Data Payload
MDP_A64_AOP4x2_QW2 - Qword A64 SIMD4x2 Atomic CMPWR Message Data Payload
MDP_A64_AOP4x2_OW1 - Oword A64 SIMD4x2 Atomic Operation Return Data Message Data Payload
MDP_A64_AOP4x2_OW2 - Oword A64 SIMD4x2 Atomic CMPWR16B Message Data Payload

Other Data Payloads

Payload
MDP_RSI - Read Surface Info Data Payload
MDP_TileW_SIMD8 - TileW SIMD8 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_OWD - A64 Oword Dual 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 Trinary Operation Message Descriptor Control Field
MDC_AOP3S - Subset Atomic Integer Trinary Operation Message Descriptor Control Field
MDC_FOP2 - Atomic Float Binary Operation Message Descriptor Control Field
MDC_FOP3 - Atomic Float Trinary 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_A32_SBSO - A32 Sideband Scale and Offset Enable Message Descriptor Control Field
MDC_A64_SBSO - A64 Sideband Scale and Offset Enable 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_MBM_CONTROL - Merged Media Block Message Header Control
MHC_MBPM_CONTROL - Pixel Masked Media Block Message Header Control
MHC_PITCH - Surface Pitch Message Header Control
MHC_PSM - Pixel Sample Mask Message Header Control
MHC_PTSS - Per Thread Scratch Space Message Header Control

Address Controls

Control
MACR_32b - SIMD 32-Bit Address Payload Control
MACR_64b - SIMD 64-Bit Address Payload Control
MACD_LOD - LOD Message Address Payload Control
MACR_LOD_SIMD8 - SIMD8 LOD Message Address Payload Control
MACD_MSAA_SN - MSAA Sample Number Message Address Control

Data Controls

Control
MDCR_DW - Dword Data Payload Register
MDCR_QW - Qword Data Payload Register
MDCR_OW - Oword Data Payload Register
MDCD_TileW - TileW SIMD8 Data Control Dword

Pixel Data Port

Cache Agents

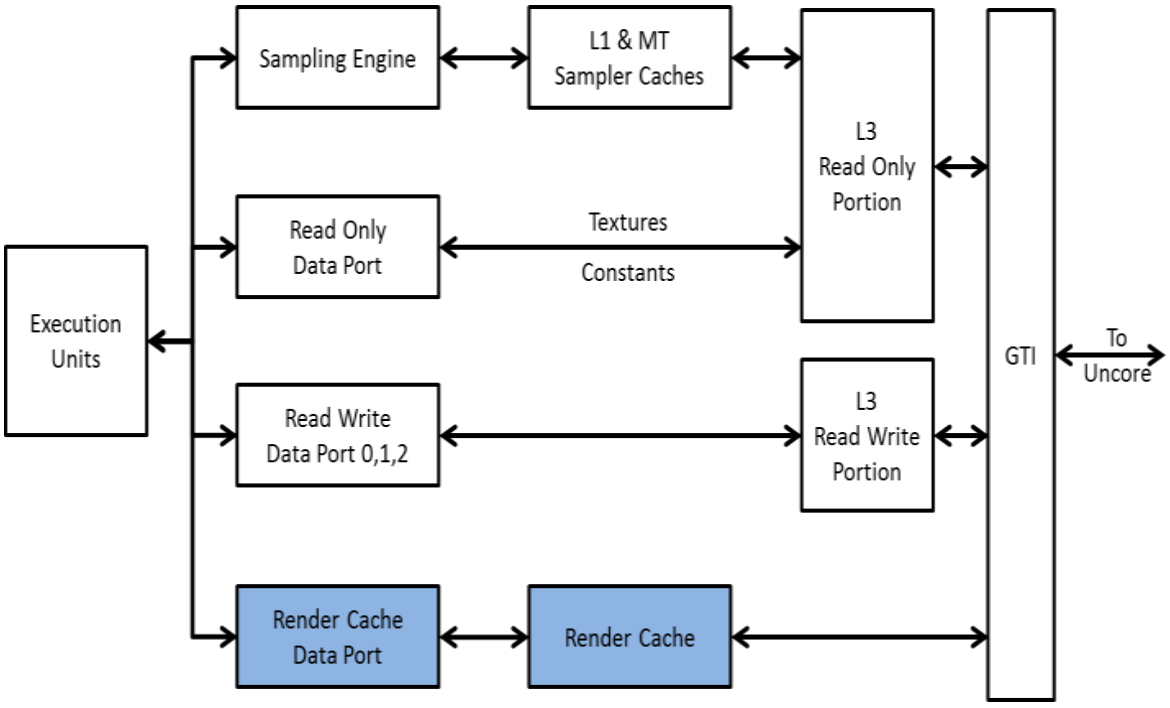
The pixel data port allows read/write accesses to render targets. Render target is cached in Render Color Cache (RCC). Auxiliary Surafce (a.k.a Control Surface) i.e. MCS buffer corresponding to a render target is cached in MSC. Messages on O-bus with Shared Function ID = Render Target (RCC) are routed to Pixel Dataport.

Shared Functions - Render Target

The Render Cache Data Port provides memory accesses for the Gen subsystem for the Render Cache. The render cache is a read/write cache that supports 3D render target surfaces.

The diagram below shows how the Render Cache Data Port connects with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.

Data Port Render Cache Connection to Memory



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.

Render Target Surfaces

This section contains information on render target surface types and surface formats.

Render Target Surface Types

All surface types, except SURFTYPE_STRBUF, 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	
R32G32B32A32_SINT	*Not Supported*	Read/Write	
R32G32B32A32_UINT	*Not Supported*	Read/Write	
R16G16B16A16_UNORM	All	Read/Write	
R16G16B16A16_SNORM	All	Read/Write	
R16G16B16A16_SINT	*Not Supported*	Read/Write	
R16G16B16A16_UINT	*Not Supported*	Read/Write	
R16G16B16A16_FLOAT	All	Read/Write	
R32G32_FLOAT	All	Read/Write	
R32G32_SINT	*Not Supported*	Read/Write	
R32G32_UINT	*Not Supported*	Read/Write	
R16G16B16X16_FLOAT	[CHV]	Read/Write	
B8G8R8A8_UNORM	All	Read/Write	
R8G8B8A8_UNORM	All	Read/Write	
R8G8B8A8_SNORM	All	Read/Write	
R8G8B8A8_SINT	*Not Supported*	Read/Write	
R8G8B8A8_UINT	*Not Supported*	Read/Write	
R16G16_UNORM	All	Read/Write	
R16G16_SNORM	All	Read/Write	
R16G16_SINT	*Not Supported*	Read/Write	
R16G16_UINT	*Not Supported*	Read/Write	
R16G16_FLOAT	All	Read/Write	
R32_SINT	*Not Supported*	Read/Write	
R32_UINT	*Not Supported*	Read/Write	
R32_FLOAT	All	Read/Write	
B8G8R8X8_UNORM		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	

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
R8G8_SNORM	All	Read/Write	
R8G8_SINT	*Not Supported*	Read/Write	
R8G8_UINT	*Not Supported*	Read/Write	
R16_UNORM	All	Read/Write	
R16_SNORM	All	Read/Write	
R16_SINT	*Not Supported*	Read/Write	
R16_UINT	*Not Supported*	Read/Write	
R16_FLOAT	All	Read/Write	
B5G5R5X1_UNORM	All	Read/Write	
B5G5R5X1_UNORM_SRGB	All	Read/Write	
A1B5G5R5_UNORM	[CHV]	Read/Write	
A4B4G4R4_UNORM	[CHV]	Read/Write	
R8_UNORM	All	Read/Write	
R8_SNORM	All	Read/Write	
R8_SINT	*Not Supported*	Read/Write	
R8_UINT	*Not Supported*	Read/Write	
A8_UNORM	All	Read/Write	
YCRCB_NORMAL	*Not Supported*	Write	
YCRCB_SWAPUVY	*Not Supported*	Write	
YCRCB_SWAPUV	*Not Supported*	Write	
YCRCB_SWAPY	*Not Supported*	Write	

Subspan/Pixel to Slot Mapping

The mapping of subspans, pixels, and samples to slots in the pixel shader dispatch depends on the number of samples and message size.

Pixels are numbered as follows within a subspan:

0 = upper left

1 = upper right

2 = lower left

3 = lower right

sspi = Starting Sample Pair Index (from the message header)

Slot Mapping

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] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		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]
SIMD16	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]
	2X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]
	4X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]
	8X	Dispatch[i]: (i=0, 2) SSPI = i Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]
	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]

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
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]

Programming Note	
Context:	Render Target Surfaces
• When SIMD32 or SIMD16 PS threads send render target writes with multiple SIMD8 and SIMD16 messages, the following must hold: • All the slots (as described above) must have a corresponding render target write irrespective of the slot's validity. A slot is considered valid when at least one sample is enabled. For example, a SIMD16 PS thread must send two SIMD8 render target writes to cover all the slots.	

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

Render Target Messages

The message operations on Render Cache Data Port use the same interface as the other data port messages. The message's Descriptor is sent to the target data port on a message sideband bus, and the other parts are sequentially transmitted between the data port and EU registers across the OBUS. The total number of registers sent and received by the data port is provided to the data port on the message sideband bus.

Descriptor	Describes the Message Type, the Message Specific Controls, and whether a Message Header is present. The Message Descriptor and the destination Data Port (SFID) are encoded as part of the *send* instruction.
Header	When present, provides additional Message specific controls. The message header is optional for some Render Target messages. When present, it is a 2-register payload.
Source Payload	Provides source data for write operations 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.

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}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_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}
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}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	{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}
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}

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_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}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_A	{Forbidden}	MDP_RTWH_A16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_M	{Forbidden}	MDP_RTWH_M16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_REP16	{Opt} MH_RT	{Forbidden}	MDP_RTWH_16REP	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_REP16	MH_RT_M	{Forbidden}	MDP_RTWH_M16REP	{Forbidden}

The sample for a pixel is killed (not written to the render target or depth buffer) if the corresponding oMask bit is zero. Bits in oMask larger than the number of multisamples are ignored.

The color payload is included if the Message Type is SIMD8 single source or SIMD8 Image Write.

The SIMD16 Replicated Data Color payload is included if the Message Type specifies single source message with replicated data. One set of R/G/B/A data is included in the message, and this data is replicated to all 16 pixels. The Replicated SIMD16 message is legal with color data and oMask, but the registers for depth, stencil, and antialias alpha data cannot be included with this message, and the corresponding bits in the message header must indicate that these registers are not present.

The following table enumerates the order that the data payload is packed in registers: Source 0 Alpha (s0A), Output Mask (oM), Pixel components (R, G, B, A), Source Depth (sZ), and Output Stencil (oS).

Summary of Data Payload Register Order for Render Target Write Messages

Message Type	Present?				Message Register												
	S0A	oM	sZ	oS	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
SIMD16	0	0	0	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A					
SIMD16	1	0	0	0	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A			
SIMD16	0	0	1	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ			
SIMD16	1	0	1	0	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ	
SIMD16	0	1	0	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A				
SIMD16	1	1	0	0	1/0soA	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A		
SIMD16	0	1	1	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ		
SIMD16	1	1	1	0	1/0s0A	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ
REP16	0	0	0	0	RGBA												
LO8DS	0	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
LO8DS	0	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
LO8DS	0	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
LO8DS	0	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
HI8DS	0	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
HI8DS	0	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
HI8DS	0	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
HI8DS	0	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
SIMD8	0	0	0	0	R	G	B	A									
SIMD8	1	0	0	0	s0A	R	G	B	A								
SIMD8	0	0	1	0	R	G	B	A	sZ								
SIMD8	1	0	1	0	s0A	R	G	B	A	sZ							
SIMD8	0	1	0	0	oM	R	G	B	A								
SIMD8	1	1	0	0	s0A	oM	R	G	B	A							
SIMD8	0	1	1	0	oM	R	G	B	A	sZ							
SIMD8	1	1	1	0	s0A	oM	R	G	B	A	sZ						
LO8DS	0	0	0	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS				
LO8DS	0	0	1	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0OoS			
LO8DS	0	1	0	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS			
LO8DS	0	1	1	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0oS		
HI8DS	0	0	0	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS				
HI8DS	0	0	1	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS			
HI8DS	0	1	0	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS			
HI8DS	0	1	1	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS		
SIMD8	0	0	0	1	R	G	B	A	oS								
SIMD8	1	0	0	1	s0A	R	G	B	A	oS							

SIMD8	0	0	1	1	R	G	B	A	sZ	oS							
SIMD8	1	0	1	1	s0A	R	G	B	A	sZ	oS						
SIMD8	0	1	0	1	oM	R	G	B	A	oS							
SIMD8	1	1	0	1	s0A	oM	R	G	B	A	oS						
SIMD8	0	1	1	1	oM	R	G	B	A	sZ	oS						
SIMD8	1	1	1	1	s0A	oM	R	G	B	A	sZ	oS					
HP SIMD16	0	0	0	0	R	G	B	A									
HP SIMD16	1	0	0	0	s0A	R	G	B	A								
HP SIMD16	0	0	1	0	R	G	B	A	sZ								
HP SIMD16	1	0	1	0	s0A	R	G	B	A	sZ							
HP SIMD16	0	1	0	0	oM	R	G	B	A								
HP SIMD16	1	1	0	0	s0A	oM	R	G	B	A							
HP SIMD16	0	1	1	0	oM	R	G	B	A	sZ							
HP SIMD16	1	1	1	0	s0A	oM	R	G	B	A	sZ						
HP REP16	0	0	0	0	RGBA												
HP LO8DS	0	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
HP LO8DS	0	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
HP LO8DS	0	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
HP LO8DS	0	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
HP HI8DS	0	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
HP HI8DS	0	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
HP HI8DS	0	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
HP HI8DS	0	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
HP SIMD8	0	0	0	0	R	G	B	A									
HP SIMD8	1	0	0	0	s0A	R	G	B	A								
HP SIMD8	0	0	1	0	R	G	B	A	sZ								
HP SIMD8	1	0	1	0	s0A	R	G	B	A	sZ							
HP SIMD8	0	1	0	0	oM	R	G	B	A								
HP SIMD8	1	1	0	0	s0A	oM	R	G	B	A							
HP SIMD8	0	1	1	0	oM	R	G	B	A	sZ							
HP SIMD8	1	1	1	0	s0A	oM	R	G	B	A	sZ						
HP LO8DS	0	0	0	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS				
HP LO8DS	0	0	1	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0OoS			
HP LO8DS	0	1	0	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS			
HP LO8DS	0	1	1	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0oS		
HP HI8DS	0	0	0	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS				
HP HI8DS	0	0	1	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS			
HP HI8DS	0	1	0	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS			
HP HI8DS	0	1	1	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS		
HP SIMD8	0	0	0	1	R	G	B	A	oS								
HP SIMD8	1	0	0	1	s0A	R	G	B	A	oS							
HP SIMD8	0	0	1	1	R	G	B	A	sZ	oS							
HP SIMD8	1	0	1	1	s0A	R	G	B	A	sZ	oS						
HP SIMD8	0	1	0	1	oM	R	G	B	A	oS							
HP SIMD8	1	1	0	1	s0A	oM	R	G	B	A	oS						
HP SIMD8	0	1	1	1	oM	R	G	B	A	sZ	oS						
HP SIMD8	1	1	1	1	s0A	oM	R	G	B	A	sZ	oS					

Programming Note	
Context:	Render Target Write Message
Typically, the last message in a pixel shader is a Render Target Write message, with EOT set.	

Programming Note	
Context:	Render Target Write Message
• The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG. • If multiple SIMD8 Dual Source messages are delivered by the pixel shader thread, each SIMD8_DUALSRC_LO message must be issued *before* the SIMD8_DUALSRC_HI message with the same Slot Group Select setting. • Output Stencil is not supported with SIMD16 Render Target Write Messages.	

Programming Note	
Context:	Render Target Writes in Multirate Shaders
• In multirate shaders, render target writes that can potentially kill pixels (via oMask, oDepth, or oStencil) cannot modify the oMask, oDepth, or oStencil value after writing to a render target. For instance if a pixel-rate render target is written, an associated sample-rate render target cannot modify these values or undefined behavior can occur. • In multirate shaders that write to render targets in multiple stages, render target 0 must be written to before any other render targets..	

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 followng 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	SIMD 8	{4}	MSD_RTR_SIMD8	MH_RT	{Forbidden}	{Forbidden}	{4} MDP_DW_SIMD16
R	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD 16	{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.

Programming Note	
Context:	Render Target Read Message
For 1d surfaces Render Target Read Messages are not supported.	

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.

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

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

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.

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.

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.

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.

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.

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.

Shared Functions Pixel Interpolater

The Pixel Interpolator provides barycentric parameters at various offsets relative to the pixel location. These barycentric parameters are in the same format and layout as those received in the pixel shader dispatch. Please refer to the “Windower” chapter in the “3D Pipeline” volume for more details on barycentric parameters.

Barycentric parameters delivered in the pixel shader payload are at pre-defined positions based on **Barycentric Interpolation Mode** bits selected in 3DSTATE_WM. The pixel interpolator allows barycentric parameters to be computed at additional locations.

Messages

The following is the message definition for the Pixel Interpolator shared function.

Programming Note	
Context:	Messages
Pixel Interpolator messages can only be delivered by pixel shader kernels.	

Execution Mask. Each bit in the execution mask enables the corresponding slot's barycentric parameter return to the destination registers.

Initiating Message

Message Descriptor

Bits	Description
19	**Header Present:** Specifies whether the message includes a header phase. Must be zero for all *Pixel Interpolator* messages. Format = Enable
18:17	Ignored
16	**SIMD Mode.** Specifies the SIMD mode of the message being sent. Format = U1 0: SIMD8 mode 1: SIMD16 mode
15	Ignored
14	**Interpolation Mode.** Specifies which interpolation mode is used. Format = U1 0: Perspective Interpolation 1: Linear Interpolation **Programming Note** **Context:** Message Descriptor This field cannot be set to "Linear Interpolation" unless **Non-Perspective Barycentric Enable** in 3DSTATE_CLIP is enabled.

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>13:12</td><td>Message Type. Specifies the type of message being sent when pixel-rate evaluation requested.
Format = U2
0: Per Message Offset (eval_snapped with immediate offset)
1: Sample Position Offset (eval_sindex)
2: Centroid Position Offset (eval_centroid)
3: Per Slot Offset (eval_snapped with register offset)
<table>
<tr><th colspan="2">Programming Note</th></tr>
<tr><th>Context:</th><td>Message Descriptor</td></tr>
<tr><td colspan="2">When Message Type is Sample Position, requesting an attribute at sample index beyond the range defined by the Forced Sample Count (aka NUM_RASTSAMPLES) is illegal.</td></tr>
</table></td></tr>
<tr><td>11</td><td>Slot Group Select. This field selects whether slots 15:0 or slots 31:16 are used for bypassed data.
Bypassed data includes the X/Y addresses and centroid position. For 8- and 16-pixel dispatches, SLOTGRP_LO must be selected on every message. For 32-pixel dispatches, this field must be set correctly for each message based on which slots are currently being processed.
0: SLOTGRP_LO: Choose bypassed data for slots 15:0.
1: SLOTGRP_HI: Choose bypassed data for slots 31:16.
<table>
<tr><th colspan="2">Programming Note</th></tr>
<tr><th>Context:</th><td>Message Descriptor</td></tr>
<tr><td colspan="2">This field must be set to SLOTGRP_LO for SIMD8 messages. SIMD8 messages always use bypassed data for slots 7:0.</td></tr>
</table></td></tr>
<tr><td>10:8</td><td>Ignored</td></tr>
<tr><td>7:0</td><td>Message Specific Control. Refer to the sections below for the definition of these bits based on Message Type.</td></tr>
</table>

"Per Message Offset" Message Descriptor

Bit	Description
7:4	**Per Message Y Pixel Offset** Specifies the Y Pixel Offset that applies to all slots. Format = S4 2's complement representing units of 1/16 pixel. Range = [-8/16, +7/16]
3:0	**Per Message X Pixel Offset** Specifies the X Pixel Offset that applies to all slots. Format = S4 2's complement representing units of 1/16 pixel. Range = [-8/16, +7/16]

"Sample Position Offset" Message Descriptor

Bits	Description
7:4	**Sample Index** Specifies the sample index that applies to all slots. Sample Index must not exceed the value of NUM_RASTSAMPLES when NUM_RASTSAMPLES > 1. From API, perspective, Forced Sample Count Defines the maximum allowable index in this message. Format = U4 Range [0, 15]
3:0	Ignored

"Centroid Position" and "Per Slot Offset" Message Descriptor

Bit	Description
7:0	Ignored

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. Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored. Range = [-8/16, +7/16]
M1.6	31:0	**Slot 6 Y Pixel Offset**

DWord	Bit	Description
M1.5	31:0	**Slot 5 Y Pixel Offset**
M1.4	31:0	**Slot 4 Y Pixel Offset**
M1.3	31:0	**Slot 3 Y Pixel Offset**
M1.2	31:0	**Slot 2 Y Pixel Offset**
M1.1	31:0	**Slot 1 Y Pixel Offset**
M1.0	31:0	**Slot 0 Y Pixel Offset**

SIMD16 Per Slot Offset Message Payload

This message payload applies only to the SIMD16 Per Slot Offset message type. The message length is 4.

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 15 X Pixel Offset**
M1.6	31:0	**Slot 14 X Pixel Offset**

DWord	Bit	Description
M1.5	31:0	**Slot 13 X Pixel Offset**
M1.4	31:0	**Slot 12 X Pixel Offset**
M1.3	31:0	**Slot 11 X Pixel Offset**
M1.2	31:0	**Slot 10 X Pixel Offset**
M1.1	31:0	**Slot 9 X Pixel Offset**
M1.0	31:0	**Slot 8 X Pixel Offset**
M2.7	31:0	**Slot 7 Y Pixel Offset** Specifies the Y pixel offset for slot 7. Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored. Range = [-8/16, +7/16]
M2.6	31:0	**Slot 6 Y Pixel Offset**
M2.5	31:0	**Slot 5 Y Pixel Offset**
M2.4	31:0	**Slot 4 Y Pixel Offset**
M2.3	31:0	**Slot 3 Y Pixel Offset**
M2.2	31:0	**Slot 2 Y Pixel Offset**
M2.1	31:0	**Slot 1 Y Pixel Offset**
M2.0	31:0	**Slot 0 Y Pixel Offset**
M3.7	31:0	**Slot 15 Y Pixel Offset**
M3.6	31:0	**Slot 14 Y Pixel Offset**
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**

DWord	Bit	Description
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**

Writeback Message

SIMD8

The response length for all SIMD8 messages is 2. The data for each slot is written only if its corresponding execution mask bit is set.

DWord	Bit	Description
W0.7	31:0	**Barycentric[1] for Slot 7** Format = IEEE_Float
W0.6	31:0	**Barycentric[1] for Slot 6**
W0.5	31:0	**Barycentric[1] for Slot 5**
W0.4	31:0	**Barycentric[1] for Slot 4**
W0.3	31:0	**Barycentric[1] for Slot 3**
W0.2	31:0	**Barycentric[1] for Slot 2**
W0.1	31:0	**Barycentric[1] for Slot 1**
W0.0	31:0	**Barycentric[1] for Slot 0**
W1.7	31:0	**Barycentric[2] for Slot 7** Format = IEEE_Float
W1.6	31:0	**Barycentric[2] for Slot 6**
W1.5	31:0	**Barycentric[2] for Slot 5**
W1.4	31:0	**Barycentric[2] for Slot 4**

DWord	Bit	Description
W1.3	31:0	**Barycentric[2] for Slot 3**
W1.2	31:0	**Barycentric[2] for Slot 2**
W1.1	31:0	**Barycentric[2] for Slot 1**
W1.0	31:0	**Barycentric[2] for Slot 0**

SIMD16

The response length for all SIMD16 messages is 4. The data for each slot is written only if its corresponding execution mask bit is set.

DWord	Bit	Description
W0.7	31:0	**Barycentric[1] for Slot 7** Format = IEEE_Float
W0.6	31:0	**Barycentric[1] for Slot 6**
W0.5	31:0	**Barycentric[1] for Slot 5**
W0.4	31:0	**Barycentric[1] for Slot 4**
W0.3	31:0	**Barycentric[1] for Slot 3**
W0.2	31:0	**Barycentric[1] for Slot 2**
W0.1	31:0	**Barycentric[1] for Slot 1**
W0.0	31:0	**Barycentric[1] for Slot 0**
W1.7	31:0	**Barycentric[2] for Slot 7** Format = IEEE_Float
W1.6	31:0	**Barycentric[2] for Slot 6**
W1.5	31:0	**Barycentric[2] for Slot 5**
W1.4	31:0	**Barycentric[2] for Slot 4**

DWord	Bit	Description
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**

Shared Functions - Unified Return Buffer (URB)

The Unified Return Buffer (URB) is a general-purpose buffer used for sending data between different threads, and, in some cases, between threads and fixed-function units (or vice-versa). A thread accesses the URB by sending messages.

URB Size

An URB entry is a logical entity within the URB, referenced by an entry handle and comprised of some number of consecutive rows. A row corresponds in size to a 256-bit EU GRF register. Read/write access to the URB is generally supported on a row-granular basis.

URB Size	URB Rows	URB Rows when SLM Enabled
See the Configurations volume.		

URB Access

The URB can be written by the following agents:

- Command Stream (CS) can write constant data into Constant URB Entries (CURBEs) as a result of processing CONSTANT_BUFFER commands.
- The Video Front End (VFE) fixed-function unit of the Media pipeline can write thread payload data in to its URB entries.
- The Vertex Fetch (VF) fixed-function unit of the 3D pipeline can write vertex data into its URB entries.
- GEN4 threads can write data into URB entries via URB_WRITE messages sent to the URB shared function.

The URB can be read by the following agents:

- The Thread Dispatcher (TD) is the main source of URB reads. As a part of spawning a thread, pipeline fixed-functions provide the TD with a number of URB handles, read offsets, and lengths. The TD reads the specified data from the URB and provide that data in the thread payload pre-loaded into GRF registers.
- The Geometry Shader (GS) and Clipper (CLIP) fixed-function units of the 3D pipeline can read selected parts of URB entries to extract vertex data required by the pipeline.
- The Windower (WM) FF unit reads back depth coefficients from URB entries written by the Strip/Fan unit.

Programming Note	
Context:	URB Access
The CPU cannot read the URB directly.	

State

The URB function is stateless, with all information required to perform a function being passed in the write message.

See URB Allocation (*Graphics Processing Engine*) for a discussion of how the URB is divided amongst the various fixed functions.

URB Messages

This section documents the global aspects of the URB messages. The actual data stored in URB entries differs for each fixed function – refer to *3D Pipeline* and the fixed-function chapters or details on 3D URB data formats and *Media* for media-specific URB data formats.

URB Handles: Unlike prior products where the URB handle contents was not specified for software use, URB handles are now specified as offsets into the URB partition in the L3 cache, in 512-bit units. Thus, kernels can now perform math operations on URB handles.

The **End of Thread** bit in the message descriptor may be set on URB messages only in threads dispatched by the vertex shader (VS), hull shader (HS), domain shader (DS), and geometry shader (GS). The **End of Thread** bit cannot be set on URB_READ* or URB_ATOMIC* messages.

Execution Mask. The low 8 bits of the execution mask on the send instruction determines which DWords from each write data phase are written or which DWords from each read phase are written to the destination GRF register. The execution mask is ignored on URB_ATOMIC* messages, because this is a scalar operation that is always enabled.

Out-of-Bounds Accesses. Reads to addresses outside of the URB region allocated in the L3 cache return 0. Writes to addresses outside of the URB region are dropped and do not modify any URB data.

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
URB Read HWORD	yes	N/A	N/A	handle + URBoffset or handle + URBoffset + offset	1, 2
URB Write HWORD	yes	N/A	N/A	handle + URBoffset or handle + URBoffset + offset	1, 2

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
URB Read OWORD	yes	N/A	N/A	handle + URBoffset or handle + URBoffset + offset	1, 2
URB Write OWORD	yes	N/A	N/A	handle + URBoffset or handle + URBoffset + offset	1, 2
URB Atomic MOV	yes	N/A	N/A	handle + URBoffset	1
URB Atomic INC	yes	N/A	N/A	handle + URBoffset	1
URB Atomic ADD	yes	N/A	N/A	handle + URBoffset	1

"offset" is in the message payload, and is per-slot.

"handle" is the handle address in the message header.

"URBoffset" is the **Global Offset** field in the URB message descriptor.

Execution Mask

The Execution Mask specified in the 'send' instruction determines which DWords within each message register are read/written to the URB.

Message Descriptor

Bit	Description
19	**Header Present** This bit must be 1 for all URB messages.
18	Ignored.
17	**Per Slot offset:** If clear, the slot offset fields in the header are ignored. If set the slot offset fields are added to the global offset to obtain the overall offset. **Programming Notes:** • This bit must be 0 for URB_ATOMIC_* messages.

Bit	Description
16	Ignored.
15	**Swizzle Control.** This field specifies which “swizzle” operation is performed on the write data. It indirectly specifies whether one or two handles are valid. 0: URB_NOSWIZZLE The message accesses a single URB entry (using **URB Handle 0**). 1: URB_INTERLEAVED The message accesses two URB entries. The data is interleaved such that the upper DWords (7:4) of each 256-bit unit contain data associated with **URB Handle 1**, and the lower DWords (3:0) contain data associated with **URB Handle 0**. **Programming Notes:** • This bit must be 0 for URB_ATOMIC_* messages.
14:4	**Global Offset.** This field specifies a destination offset (in 256-bit units) from the start of the URB entry(s), as referenced by **URB Handle** *n*, at which the data (if any) is written to or read from. When URB_INTERLEAVED is used, this field provides a 256-bit granular offset applied to both URB entries. If the **Per Slot Offset** bit is set, this offset is added to the per-slot offsets in the header to obtain the overall offset. For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units. For the URB_ATOMIC* messages, this offset is in 32-bit units instead of 256-bit units. Format = U11 Range = [0, 1023] for URB_*_HWORD messages. Range = [0, 2047] for URB_*_OWORD messages. Range = [0, 2047] for URB_ATOMIC* messages.
3:0	**URB Opcode** 0: URB_WRITE_HWORD 1: URB_WRITE_OWORD 2: URB_READ_HWORD 3: URB_READ_OWORD 4: URB_ATOMIC_MOV 5: URB_ATOMIC_INC 6: URB_ATOMIC_ADD 7: URB_SIMD8_WRITE (see URB_SIMD8_* for descriptor details)

Bit	Description
	8: URB_SIMD8_READ (see URB_SIMD8_* for descriptor details) 9-15: Reserved

URB_WRITE and URB_READ

The URB_WRITE* and URB_READ* messages share the same header definition. URB_WRITE has additional payload containing the write data, but has no writeback message. URB_READ has no payload beyond the header (message length is always one), but always has a writeback message. URB_WRITE_SIMD4x2 has a single-phase payload with the per-slot offsets followed by the write data, and has no writeback message. URB_READ_SIMD4x2 has a single phase payload containing the per-slot offsets.

Message Header

M0.5[7:0] bits in message header are used for enabling DWs in cull test, at HDC unit by HS kernel, while writing TF data using URB write messages. Cull test is performed on outside TF and HS kernel set the appropriate DW enable, which carry the TF for different domain types. When DW is enabled and if cull test is positive, HS stage will be informed by HDC unit, to cull the HS handle early at HS stage itself.

DWord	Bits	Description
M0.5	31:17	Ignored
	16	**High OWORD Enable** For URB_READ_OWORD and URB_WRITE_OWORD with NOSWIZZLE indicates whether the 128 bits of the GRF register is used. 0: 1 OWord, read into or written from the low 128 bits of the GRF register. 1: 1 OWord, read into or written from the high 128 bits of the GRF register.
	15	Vertex 1 DATA [3] / Vertex 0 DATA[7] Channel Mask When **Swizzle Control** = URB_INTERLEAVED this bit controls Vertex 1 DATA[3]. When **Swizzle Control** = URB_NOSWIZZLE this bit controls Vertex 0 DATA[7]. This bit is ANDed with the corresponding channel enable to determine the final channel enable. For the URB_READ_OWORD & URB_READ_HWORD messages, when final channel enable is 1 it indicates that Vertex 1 DATA [3] will be included in the writeback message. For the URB_WRITE_OWORD & URB_WRITE_HWORD messages, when final channel enable is 1 it indicates that Vertex 1 DATA [3] will be written to the surface. 0: Vertex 1 DATA [3] / Vertex 0 DATA[7] channel not included. 1: Vertex DATA [3] / Vertex 0 DATA[7] channel included.
	14	**Vertex 1 DATA [2] Channel Mask**
	13	**Vertex 1 DATA [1] Channel Mask**

DWord	Bits	Description
	12	**Vertex 1 DATA [0] Channel Mask**
	11	**Vertex 0 DATA [3] Channel Mask**
	10	**Vertex 0 DATA [2] Channel Mask**
	9	**Vertex 0 DATA [1] Channel Mask**
	8	**Vertex 0 DATA [0] Channel Mask**
	7:0	**Reserved**
M0.4	31:0	**Slot 1 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 256-bit units) from the start of the URB entry, as referenced by **URB Handle 1**, at which the data will be accessed. This field is ignored unless **Swizzle Control** is set to URB_INTERLEAVED. For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units. Format = U32 Range = [0, 1023] for URB_*_HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined. Range = [0, 2047] for URB_*_OWORD messages. The range of the calculated offset must fall within the range [0, 2047] or behavior is undefined.
M0.3	31:0	**Slot 0 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 256-bit units) from the start of the URB entry, as referenced by **URB Handle 0**, at which the data will be accessed. For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units. Format = U32 Range = [0, 1023] for URB_*_HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined. Range = [0, 2047] for URB_*_OWORD messages. The range of the calculated offset must fall within the range [0, 2047] or behavior is undefined.
M0.1	31:16	Reserved.
	15:14	Reserved.
	13:0	**URB Handle 1.** This is the URB handle where channel 1's results are to be written or read. This field is ignored unless **Swizzle Control** indicates interleave mode.
M0.0	31:16	Reserved.
	15:14	Reserved.
	13:0	**URB Handle 0.** This is the URB handle where channel 0's results are to be written or read.

URB_WRITE_HWORD Write Data Payload

Programming Restriction: The write data payload can be between 1 and 8 message phases long.

For the URB_WRITE_HWORD messages, the message payload will be written to the URB entries indicated by the URB return handles in the message header.

Payload	Usage
URB_NOSWIZZLE	The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The message payload contains data to be written to two separate URB entries. The payload data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.

URB_NOSWIZZLE

URB_NOSWIZZLE is used to simply write data into consecutive URB locations (no data swizzling applied).

Programming Note	
Context:	URB_NOSWIZZLE
The URB function will use (not ignore) the Channel Enables associated with this message.	

When URB_NOSWIZZLE is used to write vertex data, the following table shows an example layout of a URB_NOSWIZZLE payload containing one (non-interleaved) vertex containing *n* pairs of 4-DWord vertex elements (where for the example, *n* is >2).

DWord	Bit	Description
M1.7	31:0	**Vertex Data [7]**
M1.6	31:0	**Vertex Data [6]**
M1.5	31:0	**Vertex Data [5]**
M1.4	31:0	**Vertex Data [4]**
M1.3	31:0	**Vertex Data [3]**
M1.2	31:0	**Vertex Data [2]**
M1.1	31:0	**Vertex Data [1]**
M1.0	31:0	**Vertex Data [0]**
M2.7	31:0	**Vertex Data [15]**
M2.6	31:0	**Vertex Data [14]**

DWord	Bit	Description
M2.5	31:0	**Vertex Data [13]**
M2.4	31:0	**Vertex Data [12]**
M2.3	31:0	**Vertex Data [11]**
M2.2	31:0	**Vertex Data [10]**
M2.1	31:0	**Vertex Data [9]**
M2.0	31:0	**Vertex Data [8]**
…		**…**
Mn.7	31:0	**Vertex Data [8(n-1)+7]**
Mn.6	31:0	**Vertex Data [8(n-1)+6]**
Mn.5	31:0	**Vertex Data [8(n-1)+5]**
Mn.4	31:0	**Vertex Data [8(n-1)+4]**
Mn.3	31:0	**Vertex Data [8(n-1)+3]**
Mn.2	31:0	**Vertex Data [8(n-1)+2]**
Mn.1	31:0	**Vertex Data [8(n-1)+1]**
Mn.0	31:0	**Vertex Data [8(n-1)+0]**

URB_INTERLEAVED

The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing *n* 4-DWord vertex elements (n>1).

Programming Note	
Context:	URB_INTERLEAVED
• The URB function will use (not ignore) the Channel Enables associated with this message. • Writes to overlapping addresses of vertex0 and vertex1 will have undefined write ordering.	

DWord	Bit	Description
M1.7	31:0	**Vertex 1 Data [3]**
M1.6	31:0	**Vertex 1 Data [2]**
M1.5	31:0	**Vertex 1 Data [1]**
M1.4	31:0	**Vertex 1 Data [0]**
M1.3	31:0	**Vertex 0 Data [3]**
M1.2	31:0	**Vertex 0 Data [2]**
M1.1	31:0	**Vertex 0 Data [1]**
M1.0	31:0	**Vertex 0 Data [0]**
M2.7	31:0	**Vertex 1 Data [7]**
M2.6	31:0	**Vertex 1 Data [6]**
M2.5	31:0	**Vertex 1 Data [5]**
M2.4	31:0	**Vertex 1 Data [4]**
M2.3	31:0	**Vertex 0 Data [7]**
M2.2	31:0	**Vertex 0 Data [6]**
M2.1	31:0	**Vertex 0 Data [5]**
M2.0	31:0	**Vertex 0 Data [4]**
...		**...**
Mn.7	31:0	**Vertex 1 Data [4(n-1)+3]**
Mn.6	31:0	**Vertex 1 Data [4(n-1)+2]**
Mn.5	31:0	**Vertex 1 Data [4(n-1)+1]**

DWord	Bit	Description
Mn.4	31:0	**Vertex 1 Data [4(n-1)+0]**
Mn.3	31:0	**Vertex 0 Data [4(n-1)+3]**
Mn.2	31:0	**Vertex 0 Data [4(n-1)+2]**
Mn.1	31:0	**Vertex 0 Data [4(n-1)+1]**
Mn.0	31:0	**Vertex 0 Data [4(n-1)+0]**

URB_READ_HWORD Writeback Message

For the URB_READ_HWORD messages, the URB entries indicated by the URB handles in the message header are read and returned in the writeback message. The amount of read data returned is determined by the **Response Length** field.

Programming Restriction: The writeback message can be between 1 and 8 message phases long.

While GS threads will read one vertex at a time to the URB, the VS will read two interleaved vertices. The description of the URB read messages will refer to the per-vertex DWords described in the Vertex URB Entry Formats section of the *3D Overview* chapter.

Payload	Usage
URB_NOSWIZZLE	The writeback message contains data read from a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The writeback message contains data read from two separate URB entries. The data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.

URB_NOSWIZZLE

URB_NOSWIZZLE is used to simply read data into consecutive URB locations (no data interleaving applied).

When URB_NOSWIZZLE is used to read vertex data, the following table shows an example layout of a URB_NOSWIZZLE writeback message containing one (non-interleaved) vertex containing *n* pairs of 4-DWord vertex elements (where for the example, *n* is >2).

DWord	Bit	Description
W0.7	31:0	**Vertex Data [7]**
W0.6	31:0	**Vertex Data [6]**

DWord	Bit	Description
W0.5	31:0	**Vertex Data [5]**
W0.4	31:0	**Vertex Data [4]**
W0.3	31:0	**Vertex Data [3]**
W0.2	31:0	**Vertex Data [2]**
W0.1	31:0	**Vertex Data [1]**
W0.0	31:0	**Vertex Data [0]**
W1.7	31:0	**Vertex Data [15]**
W1.6	31:0	**Vertex Data [14]**
W1.5	31:0	**Vertex Data [13]**
W1.4	31:0	**Vertex Data [12]**
W1.3	31:0	**Vertex Data [11]**
W1.2	31:0	**Vertex Data [10]**
W1.1	31:0	**Vertex Data [9]**
W1.0	31:0	**Vertex Data [8]**
...		**...**
Wn.7	31:0	**Vertex Data [8n+7]**
Wn.6	31:0	**Vertex Data [8n+6]**
Wn.5	31:0	**Vertex Data [8n+5]**
Wn.4	31:0	**Vertex Data [8n+4]**
Wn.3	31:0	**Vertex Data [8n+3]**
Wn.2	31:0	**Vertex Data [8n+2]**

DWord	Bit	Description
Wn.1	31:0	**Vertex Data [8n+1]**
Wn.0	31:0	**Vertex Data [8n+0]**

URB_INTERLEAVED

The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing *n* 4-DWord vertex elements (n>1).

DWord	Bit	Description
W0.7	31:0	**Vertex 1 Data [3]**
W0.6	31:0	**Vertex 1 Data [2]**
W0.5	31:0	**Vertex 1 Data [1]**
W0.4	31:0	**Vertex 1 Data [0]**
W0.3	31:0	**Vertex 0 Data [3]**
W0.2	31:0	**Vertex 0 Data [2]**
W0.1	31:0	**Vertex 0 Data [1]**
W0.0	31:0	**Vertex 0 Data [0]**
W1.7	31:0	**Vertex 1 Data [7]**
W1.6	31:0	**Vertex 1 Data [6]**
W1.5	31:0	**Vertex 1 Data [5]**
W1.4	31:0	**Vertex 1 Data [4]**
W1.3	31:0	**Vertex 0 Data [7]**
W1.2	31:0	**Vertex 0 Data [6]**
W1.1	31:0	**Vertex 0 Data [5]**
W1.0	31:0	**Vertex 0 Data [4]**

DWord	Bit	Description
…		…
Wn.7	31:0	**Vertex 1 Data [4n+3]**
Wn.6	31:0	**Vertex 1 Data [4n+2]**
Wn.5	31:0	**Vertex 1 Data [4n+1]**
Wn.4	31:0	**Vertex 1 Data [4n+0]**
Wn.3	31:0	**Vertex 0 Data [4n+3]**
Wn.2	31:0	**Vertex 0 Data [4n+2]**
Wn.1	31:0	**Vertex 0 Data [4n+1]**
Wn.0	31:0	**Vertex 0 Data [4n+0]**

URB_WRITE_OWORD Write Data Payload

For the URB_WRITE_OWORD messages, the message payload will be written to the URB entries indicated by the URB return handles in the message header.

Payload	Usage
URB_NOSWIZZLE	The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The message payload contains data to be written to two separate URB entries. The payload data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.

URB_NOSWIZZLE

URB_NOSWIZZLE is used to simply write data into a single 128-bit URB location (no data swizzling applied).

Programming Note	
Context:	URB_NOSWIZZLE
The URB function will use (not ignore) the Channel Enables associated with this message.	

When URB_NOSWIZZLE is used to write vertex data, the following table shows an example layout of a URB_NOSWIZZLE payload containing one (non-interleaved) vertex containing 4-DWord vertex elements and HIGH OWORD ENABLE is 0.

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored
M1.4	31:0	Ignored
M1.3	31:0	**Vertex 0 Data [3]**
M1.2	31:0	**Vertex 0 Data [2]**
M1.1	31:0	**Vertex 0 Data [1]**
M1.0	31:0	**Vertex 0 Data [0]**

When URB_NOSWIZZLE is used to write vertex data, the following table shows an example layout of a URB_NOSWIZZLE payload containing one (non-interleaved) vertex containing 4-DWord vertex elements and HIGH OWORD ENABLE is 1.

DWord	Bit	Description
M1.7	31:0	**Vertex 0 Data [3]**
M1.6	31:0	**Vertex 0 Data [2]**
M1.5	31:0	**Vertex 0 Data [1]**
M1.4	31:0	**Vertex 0 Data [0]**
M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	Ignored

URB_INTERLEAVED

The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing 4-DWord vertex elements.

Programming Note	
Context:	URB_INTERLEAVED
• The URB function will use (not ignore) the Channel Enables associated with this message. • Writes to overlapping addresses of vertex0 and vertex1 will have undefined write ordering.	

DWord	Bit	Description
M1.7	31:0	**Vertex 1 Data [3]**
M1.6	31:0	**Vertex 1 Data [2]**
M1.5	31:0	**Vertex 1 Data [1]**
M1.4	31:0	**Vertex 1 Data [0]**
M1.3	31:0	**Vertex 0 Data [3]**
M1.2	31:0	**Vertex 0 Data [2]**
M1.1	31:0	**Vertex 0 Data [1]**
M1.0	31:0	**Vertex 0 Data [0]**

URB_READ_OWORD Writeback Message

For the URB_READ_HWORD messages, the URB entries indicated by the URB handles in the message header are read and returned in the writeback message. The amount of read data returned is determined by the **Response Length** field.

Programming Note	
Context:	URB_READ_OWORD Writeback Message
Response Length must be set to 1.	

While GS threads will read one vertex at a time to the URB, the VS will read two interleaved vertices. The description of the URB read messages will refer to the per-vertex DWords described in the Vertex URB Entry Formats section of the *3D Overview* chapter.

Payload	Usage
URB_NOSWIZZLE	The writeback message contains data read from a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The writeback message contains data read from two separate URB entries. The data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.

URB_NOSWIZZLE

URB_NOSWIZZLE is used to simply read data into consecutive URB locations (no data interleaving applied).

When URB_NOSWIZZLE is used to read vertex data, the following table shows an example layout of a URB_NOSWIZZLE writeback message containing one (non-interleaved) vertex containing 4-DWord vertex elements and HIGH OWORD ENABLE is 0.

DWord	Bit	Description
W0.7	31:0	Reserved (not written to GRF)
W0.6	31:0	Reserved (not written to GRF)
W0.5	31:0	Reserved (not written to GRF)
W0.4	31:0	Reserved (not written to GRF)
W0.3	31:0	**Vertex Data [3]**
W0.2	31:0	**Vertex Data [2]**
W0.1	31:0	**Vertex Data [1]**
W0.0	31:0	**Vertex Data [0]**

When URB_NOSWIZZLE is used to read vertex data, the following table shows an example layout of a URB_NOSWIZZLE writeback message containing one (non-interleaved) vertex containing 4-DWord vertex elements and HIGH OWORD ENABLE is 1.

DWord	Bit	Description
W0.7	31:0	**Vertex Data [3]**
W0.6	31:0	**Vertex Data [2]**

DWord	Bit	Description
W0.5	31:0	**Vertex Data [1]**
W0.4	31:0	**Vertex Data [0]**
W0.3	31:0	Reserved (not written to GRF)
W0.2	31:0	Reserved (not written to GRF)
W0.1	31:0	Reserved (not written to GRF)
W0.0	31:0	Reserved (not written to GRF)

URB_INTERLEAVED

The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing 4-DWord vertex elements.

DWord	Bit	Description
W0.7	31:0	**Vertex 1 Data [3]**
W0.6	31:0	**Vertex 1 Data [2]**
W0.5	31:0	**Vertex 1 Data [1]**
W0.4	31:0	**Vertex 1 Data [0]**
W0.3	31:0	**Vertex 0 Data [3]**
W0.2	31:0	**Vertex 0 Data [2]**
W0.1	31:0	**Vertex 0 Data [1]**
W0.0	31:0	**Vertex 0 Data [0]**

URB_ATOMIC

The URB_ATOMIC messages implement atomic operations on a single DWord in the URB. The location of the DWord within the URB is specified by the single URB handle and the **Global Offset** field in the message descriptor, which for these messages is a DWord offset from the URB handle. The DWord selected is operated on according to the following table:

URB Opcode	new_dst	ret
URB_ATOMIC_MOV	0	none
URB_ATOMIC_INC	old_dst + 1	old_dst

URB Opcode	new_dst	ret
URB_ATOMIC_ADD	old_dst + src0	old_dst

The previous contents of the DWord are returned in the destination register for operations that update the DWord value, such as URB_ATOMIC_INC. The URB_ATOMIC_MOV opcode does not return data (response length must be zero).

The URB_ATOMIC* messages consist only of the header. A single URB handle is specified.

Message Header

DWord	Bits	Description
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	**Source0 Data** Specifies the source 0 data for the atomic operation. This field is ignored for the URB_ATOMIC_INC message. Format = U32
M0.1	31:0	Ignored
M0.0	31:16	Ignored
	15:0	**URB Handle.** The URB handle to access.

Writeback Message

A writeback message is only returned for URB atomic operations that update the DWord value, such as URB_ATOMIC_INC. Only the low 32 bits of the destination GRF register are overwritten with the return data.

DWord	Bits	Description
W0.7:1		Reserved (not written to GRF)
W0.0	31:0	**Return Data** Specifies the value of the return data for the atomic operation. Format = U32

URB_SIMD8_Write and URB_SIMD8_Read

Programming Note	
Context:	URB_SIMD8_Write and URB_SIMD8_Read
The constant, sampler, and render caches are always non-coherent.	

Message Descriptor

Bit	Description
19	**Header Present** This bit must be set to **one** for all URB messages.
18	Ignored
17	**Per Slot offset Present:** If clear, then slot offset message phase is absent. If set then slot offset message phase is present and the per slot offsets are added to the global offset to obtain the overall offset.
16	**Ignored**
15	**Channel Mask Present:** If clear then the channel Mask Message phase is not present. If set then the channel mask message phase is present and will be used to mask data elements read or written.
14:4	**Global Offset.** This field specifies a destination offset (in 128-bit units) from the start of the URB entry(s), as referenced by **URB Handle** *n*, at which the data (if any) will be written to or read from. If the **Per Slot Offset** bit is set, this offset is added to the per-slot offsets in the header to obtain the overall offset. Format = U11 Range = [0, 2047]
3:0	**URB Opcode** 0: URB_WRITE_HWORD 1: URB_WRITE_OWORD 2: URB_READ_HWORD 3: URB_READ_OWORD 4: URB_ATOMIC_MOV 5: URB_ATOMIC_INC

Bit	Description
	6: URB_ATOMIC_ADD 7: URB_SIMD8_WRITE 8: URB_SIMD8_READ 9-15: Reserved

Message Header

DWord	Bits	Description
M0.7	31:16	Reserved
M0.7	15:14	Reserved
M0.7	13:0	**URB Handle 7.** This is the URB handle where channel 7's results are written or read.
M0.6	31:16	Reserved
M0.6	15:14	Reserved
M0.6	13:0	**URB Handle 6.** This is the URB handle where channel 6's results are written or read.
M0.5	31:16	Reserved
M0.5	15:14	Reserved
M0.5	13:0	**URB Handle 5.** This is the URB handle where channel 5's results are written or read.
M0.4	31:16	Reserved
M0.4	15:14	Reserved
M0.4	13:0	**URB Handle 4.** This is the URB handle where channel 4's results are written or read.
M0.3	31:16	Reserved
M0.3	15:14	Reserved
M0.3	13:0	**URB Handle 3.** This is the URB handle where channel 3's results are written or read.
M0.2	31:16	Reserved
M0.2	15:14	Reserved
M0.2	13:0	**URB Handle 2.** This is the URB handle where channel 2's results are written or read.
M0.1	31:16	Reserved
M0.1	15:14	Reserved
M0.1	13:0	**URB Handle 1.** This is the URB handle where channel 1's results are written or read.
M0.0	31:16	Reserved
M0.0	15:14	Reserved
M0.0	13:0	**URB Handle 0.** This is the URB handle where channel 0's results are written or read.

Per Slot Offset Message Phase

When the **Per Slot offset Present** bit in the descriptor is set then the Per slot offset message phase is sent by the EUs. The per slot message phase occurs immediately after the header.

DWord	Bit	Description
M1.7	31:0	**Slot 7 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 7**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.6	31:0	**Slot 6 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 6**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.5	31:0	**Slot 5 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 5**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.4	31:0	**Slot 4 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 4**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.3	31:0	**Slot 3 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 3**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.2	31:0	**Slot 2 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 2**, at which the data will be accessed. Format = U32 Range = [0, 2047]
M1.1	31:0	**Slot 1 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 1**, at which

DWord	Bit	Description
		the data will be accessed. Format = U32 Range = [0, 2047]
M1.0	31:0	**Slot 0 Offset.** This field, after adding to the **Global Offset** field in the message descriptor, specifies the offset (in 128-bit units) from the start of the URB entry, as referenced by **URB Handle 0**, at which the data will be accessed. Format = U32 Range = [0, 2047]

Channel Mask Message Phase

When the **Channel Mask Present** bit in the descriptor is set then the channel mask message phase is sent by the EUs. The channel mask message phase occurs after the per slot message phase if the per slot message phase exists else it occurs after the header.

DWord	Bit	Description
M2.7	31:24	**Reserved.**
	23	Vertex 7 DATA [7] Channel Mask This bit is ANDed with the corresponding channel enable to determine the final channel enable. For the URB_SIMD8_READ messages, when final channel enable is 1 it indicates that Vertex 7 DATA [7] will be included in the writeback message. For the URB_SIMD8_WRITE messages, when final channel enable is 1 it indicates that Vertex 7 DATA [7] will be written to the surface. 0: Vertex 7 DATA [7] channel not included 1: Vertex 7 DATA [7] channel included
	22	**Vertex 7 DATA [6] Channel Mask**
	21	**Vertex 7 DATA [5] Channel Mask**
	20	**Vertex 7 DATA [4] Channel Mask**
	19	**Vertex 7 DATA [3] Channel Mask**
	18	**Vertex 7 DATA [2] Channel Mask**
	17	**Vertex 7 DATA [1] Channel Mask**

DWord	Bit	Description
	16	**Vertex 7 DATA [0] Channel Mask**
	15:0	**Reserved.**
M2.6	31:24	**Reserved.**
	23:16	**Vertex 6 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.5	31:24	**Reserved.**
	23:16	**Vertex 5 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.4	31:24	**Reserved.**
	23:16	**Vertex 4 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.3	31:24	**Reserved.**
	23:16	**Vertex 3 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.2	31:24	**Reserved.**
	23:16	**Vertex 2 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.1	31:24	**Reserved.**
	23:16	**Vertex 1 DATA [7:0] Channel Mask**
	15:0	**Reserved.**
M2.0	31:24	**Reserved.**

DWord	Bit	Description
	23:16	**Vertex 0 DATA [7:0] Channel Mask**
	15:0	**Reserved.**

Workarounds

Issue
URB SIMD8 Channel Mask- URB SIMD8 Messages not supporting combinations of mixed EU settings
URB SIMD8 Channel Mask-Address corruption - URB SIMD8 Channel Mask-Address corruption

Write Data Payload

The write data payload can be between 1 and 8 message phases long.

DWord	Bit	Description
M3.7	31:0	**Vertex 7 DATA [0]**
M3.6	31:0	**Vertex 6 DATA [0]**
M3.5	31:0	**Vertex 5 DATA [0]**
M3.4	31:0	**Vertex 4 DATA [0]**
M3.3	31:0	**Vertex 3 DATA [0]**
M3.2	31:0	**Vertex 2 DATA [0]**
M3.1	31:0	**Vertex 1 DATA [0]**
M3.0	31:0	**Vertex 0 DATA [0]**
...	...	**...**
M10.7	31:0	**Vertex 7 DATA [7]**
M10.6	31:0	**Vertex 6 DATA [7]**
M10.5	31:0	**Vertex 5 DATA [7]**
M10.4	31:0	**Vertex 4 DATA [7]**

DWord	Bit	Description
M10.3	31:0	**Vertex 3 DATA [7]**
M10.2	31:0	**Vertex 2 DATA [7]**
M10.1	31:0	**Vertex 1 DATA [7]**
M10.0	31:0	**Vertex 0 DATA [7]**

Writeback Message

The writeback message can be between 1 and 8 message phases long.

DWord	Bit	Description
M0.7	31:0	**Vertex 7 DATA [0]**
M0.6	31:0	**Vertex 6 DATA [0]**
M0.5	31:0	**Vertex 5 DATA [0]**
M0.4	31:0	**Vertex 4 DATA [0]**
M0.3	31:0	**Vertex 3 DATA [0]**
M0.2	31:0	**Vertex 2 DATA [0]**
M0.1	31:0	**Vertex 1 DATA [0]**
M0.0	31:0	**Vertex 0 DATA [0]**
...	...	**...**
M7.7	31:0	**Vertex 7 DATA [7]**
M7.6	31:0	**Vertex 6 DATA [7]**
M7.5	31:0	**Vertex 5 DATA [7]**
M7.4	31:0	**Vertex 4 DATA [7]**
M7.3	31:0	**Vertex 3 DATA [7]**

DWord	Bit	Description
M7.2	31:0	**Vertex 2 DATA [7]**
M7.1	31:0	**Vertex 1 DATA [7]**
M7.0	31:0	**Vertex 0 DATA [7]**

Message Gateway

The Message Gateway has these functions:

- Barrier messages for thread-to-thread synchronization
- Timestamp provide a counter with a fixed frequency not dependent on the core clock.
- End of Thread messages (sent to Thread Spawner), to exit the GPGPU and Media threads.

Message Descriptor

The following message descriptor applies to all messages supported by Message Gateway.

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>19</td><td>Header Present. This bit must be 0 for all Message Gateway messages.</td></tr>
<tr><td>18:17</td><td>Ignored.</td></tr>
<tr><td>16:3</td><td>Reserved: MBZ</td></tr>
<tr><td>2:0</td><td>SubFuncID. Identify the supported sub-functions by Message Gateway. Encodings are:
<table>
<tr><th>Value</th><th>Name</th><th>Description</th></tr>
<tr><td>011b</td><td>GetTimeStamp</td><td>Read absolute and relative timestamps.</td></tr>
<tr><td>100b</td><td>BarrierMsg</td><td>Record an additional thread reaching the barrier</td></tr>
</table>
All other encodings Reserved.</td></tr>
</table>

GetTimeStamp Message

The GetTimeStamp message gives the ability for a requester thread to read the timestamps back from the message gateway. The message consists of a single 256-bit message payload.

AbsoluteTimeLap is based on an absolute wall clock in unit of nSec/uSec that is independent of context switch or GPU frequency adjustment. Message Gateway shares the same GPU timestamp. Details can be found in the TIMESTAMP register section in *vol1c Memory Interface and Command Stream*.

RelativeTimeLap is based on a relative time count that is counting the GPU clocks for the context. The relative time count is saved/restored during context switch.

Message Payload

DWord	Bits	Description
M0.5	31	**Return to High GRF**: 0: The return 128-bit data goes to the first half of the destination GRF register. 1: The return 128-bit data goes to the second half of the destination GRF register.
	30:8	Reserved: MBZ
	7:0	**Dispatch ID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. This field is ignored by Message Gateway. This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
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

Writeback Message to Requester Thread

As the writeback message is only sent if the **AckReq** bit in the message descriptor is set, **AckReq** bit must be set for this message.

Only half of the destination GRF register is updated (via write-enables). The other half of the register is not changed. This is determined by the **Return to High GRF** control field.

Writeback Message if Return to High GRF is set to 0:

DWord	Bit	Description
W0.7:4		Reserved (not overwritten)
W0.3	31:0	**RelativeTimeLapHigh:** This field returns the MSBs of time lap for the relative clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12

DWord	Bit	Description
W0.2	31:20	**RelativeTimeLapLow:** This field returns the LSBs of time lap for the relative clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ
W0.1	31:0	**AbsoluteTimeLapHigh:** This field returns the MSBs of time lap for the absolute clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
W0.0	31:20	**AbsoluteTimeLapLow:** This field returns the LSBs of time lap for the absolute clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ

Writeback Message if Return to High GRF is set to 1:

DWord	Bit	Description
W0.7	31:0	**RelativeTimeLapHigh**
W0.6	31:20	**RelativeTimeLapLow**
	19:0	Reserved : MBZ
W0.5	31:0	**AbsoluteTimeLapHigh**
W0.4	31:20	**AbsoluteTimeLapLow**
	19:0	Reserved : MBZ
W0.3:0		Reserved : MBZ

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
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.	

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	**Barrier ID MSB.** This field is bit[4] of the BarrierID, for full 5-bit barrier ID it should be combined with Barrier ID[3:0]. Format: U1
	30	**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.
	27:24	**BarrierID.** This field indicates which one from the 16 Barrier States is updated. There is a fifth bit for the BarrierID, in bit 31. Format: U4 Note: This field location matches with that of R0 header.
	23:16	**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. Do not use this control if the barrier is linked with other barriers in other subslices. Format: Enable
	14:9	**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.

DWord	Bits	Description
	8:0	**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: *Successful*. 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.

Media Sampler

This section describes the functionality of the Media Sampler.

Shared Functions – Video Motion Estimation

The Video Motion Estimation (VME) engine is a shared function that provides motion estimation services. It includes motion estimation for various block sizes and also standard specific operations such as

- Motion estimation and mode decision for AVC
- Intra prediction and mode decision for AVC
- Motion estimation and mode decision for MPEG2
- Motion estimation and mode decision for VC1

The motion estimation engine may also be used for other coding standards or other video processing applications.

Theory of Operation

VME performs a sequence of operations to find the best mode for a given macroblock. Each operation step can be enabled/disabled through the control of the income message. Early termination, skipping of subsequent operation steps, is also supported when certain search criteria are met.

VME contains the following operation steps:

1. Skip check
2. IME: Integer motion estimation
3. FME: Fractional motion estimation
4. BME: Bidirectional motion estimation
5. IPE: Intra prediction estimation (AVC only)

Shape Decision

As a terminology, we call sub-block shapes: 8x4, 4x8, and 4x4 *minor shapes* (corresponding to *sub-partitions* of 8x8 sub-macroblock), and 16x16, 16x8, 8x16, and 8x8 *major shapes* (corresponding to *sub-macroblocks* of a 16x16 macroblock)**.**

If the maximal allowed number of motion vectors **MaxNumMVs** (**MaxNumMVs** = **MaxNumMVsMinusOne** + 1) is less than 4, we will set minor MV flag off: **MinorMVsFlag** = 0, *i.e.* no minor motion vectors will be generated.

The reason of having this parameter **MaxNumMVs** is due to high level AVC conformance restrictions for certain profiles: *the total number of motion vectors of any two consecutive macroblocks not exceeding 16 (or 32)*. The mechanism here allows a reasonable degree of user control. In disable cases, **MaxNumMVs** should be set to 32.

In the coding process of VME, the shape decision is done in multiple locations:

1. After IME and before FME, intermediate shape decision is performed to reduce the FME searching candidates
2. After FME and before BME, existing shape decision is revised among the remaining candidates and to see if there is further reduction.
3. Final shape decision is done after BME.

Partition decision before BME uses unidirectional motion vector count to meet **MaxNumMVs** requirement. Adding BME for the partition candidates may exceed **MaxNumMVs**. As BME is performed on a block by block basis using the block order for a given partition, BME step for a given block is skipped and the best unidirectional motion vectors are used for the block if the overall motion vector count exceeds **MaxNumMVs** when that particular block is switched to bidirectional. The process continues to the last block of the partition.

Note: This is a sub-optimal solution to simplify the hardware implementation. For some cases, bidirectional modes with larger sub-partitions might be better than unidirectional modes with finer sub-partitions.

The VME implementation has the following restriction: Multiple partition candidates are only enabled if **PartCandidateEn** is set. And this only applies to source block of size 16x16.

If **PartCandidateEn** is not set, only the best partition is kept in state 1 (after IME) above and carried through FME and BME. In other words, FME if enabled only operates on one partition candidate, and BME if enabled only operates on one partition candidate. Bidirectional mode check only applies to the partition candidates that meet the bidirectional restriction provided by **BiSubMbPartMask**. For example, if a minor partition determined based on best unidirectional cost function is not 8x8 but one of 4x8, 8x4 or 4x4, VME skips the bidirectional mode check.

If **PartCandidateEn** is set, up to two sets of candidates are maintained by VME hardware, if the second best partition candidate is within **PartToleranceThrhd** from the best one. The second best partition is selected only from the two major partition candidates based on the unidirectional motion vector count, subject to that the major partition is enabled:

- 1MV: The 16x16 partition
- 4MV: The 4x(8x8) partition with no minor shape

The following partitions are not supported as alternative partition.

- 2MV: The best of 2x(16x8) and 2x(8x16) partitions
- More than 4MV: The best of all 4x(8x8) partitions with at least one 8x8 having minor shape of 8x4, 4x8 or 4x4

Major Shape Decision Prior to FME

Now considering the best of each 8x8 is done, and we have the total cost-adjusted-distortion for this sub-block level partition. Now among the four choices: the resulting 8x8 sub-partitioning, one 16x16, two 16x8, and two 8x16, the one gives the best cost-adjusted-distortion, will determine the final decision of partitioning shape. Any among these four, if its cost-adjusted-distortion is within *the intermediate*

tolerance (which is a predefined system state) from the best distortion will be marked as **candidate shape**s.

Notice that, when the intermediate tolerance is set to 0, only the best shape will be selected as the candidate. When the intermediate tolerance is large, all four shapes will become candidates.

Assume we have all the distortions for majors enumerated in DistoMajor[k], where k = 0, 1, 2, 3, 4, and 5, for 16x16, 16x8, 8x16, the combined minors, 16x8 field, and 8x8 field respectively. Assume BestDisto is equal to the minimal of the six values DistoMajor[k], for k = 0, ...5. Assume the intermediate tolerance is IntTol, the major shape k is a candidate shape if and only if DistoMajor[k]<=BestDisto+IntTol.

Shape Update after FME

Among all the candidate shapes, we recheck the distortion, if any of them is no longer with in the intermediate tolerance **DistortionTolerance** from the best choice; we drop it for reduced calculation.

Final Code Decision after BME

For any given candidate shape, for each motion vector, if we do have improved distortion by switch from the single direction to bi-direction, then we do it, unless the increased number of motion vectors hits above **MaxNumMVs**; in this case, we take as many as possible first the ones generate the most improvement.

Then, we choose the best among the improved candidate shapes.

Surfaces

The data elements accessed by VME are called “surfaces”. Surfaces are accessed using the surface state model.

VME uses the binding table to bind indices to surface state, using the same mechanism used by the sampling engine. A **Binding Table Index** (specified in the message descriptor) of less than 255 is used to index into the binding table, and the binding table entry contains a pointer to the SURFACE_STATE. SURFACE_STATE contains the parameters defining the surface to be accessed, including its location, format, and size.

State

BINDING_TABLE_STATE

VME uses the binding table to retrieve surface state. Refer to *Sampling Engine* for the definition of this state.

SURFACE_STATE

VME uses the surface state for current and reference surfaces. Refer to *Sampling Engine* for the definition of this state.

VME_STATE

This state structure contains the state used by the VME engine for data processing. VME state contains the motion search path location tables and rate-distortion weight look-up-tables. As the two sets of tables are fairly large, they are accessed as two separate states via state indexing mechanism so that applications can inter-mix the use of the search path tables and RDLUT tables.

Even though VME engine has its unique shared function ID (see Target Function ID field in the SEND instruction), the VME state is delivered through the Sampler State Pointer. When the General Purpose Pipe is used, the **Sampler State Pointer** is programmed in the MEDIA_INTERFACE_DESCRIPTOR_LOAD command and delivered directly to Sampler/VME by hardware. This posts one usage limitation. As the VME state is overloaded on top of the Sampler State Pointer, VME messages cannot be intermixed with other Sampler messages.

Each VME state may contain up to 8 VME_SEARCH_PATH_LUT_STATE. When multiple VME_SEARCH_PATH_LUT_STATE are used, they need to be stored in memory contiguously. Each VME_SEARCH_PATH_LUT_STATE contains 32 dwords in comparison of 4 dwords of a Sampler State. When enabling sampler state pre-fetch (programming the **Sampler Count** field in the MEDIA_INTERFACE_DESCRIPTOR_LOAD command), one VME_SEARCH_PATH_LUT_STATE is equivalent to 8 Samplers. Hardware may support up to two VME_SEARCH_PATH_LUT_STATE to be pre-fetched (See See 3D_Media_GPGPU chapter, Media_GPGPU_Pipeline for more details).

VME_SEARCH_PATH_LUT_STATE

Up to eight VME_SEARCH_PATH_LUT_STATE allowed for a message to select. Each state contains one set of search path locations, and four sets of rate distortion cost function LUT for various modes and rate distortion cost function LUT for motion vectors (relative to 'cost center'). Motion vector cost function is provided as a piece-wise-linear curve with only the values of the power-of-2 positions provided.

DWord	Bit	Description
0:13		**Search Path**
0	31:24	Search Path Location [3] (X, Y) – Relative distance from location [2]
	23:16	Search Path Location [2] (X, Y) – Relative distance from location [1]
	15:8	Search Path Location [1] (X, Y) – Relative distance from location [0]
	7:4	Search Path location [0] (Y) – specifies relative Y distance of the **next walk** from the starting position in unit of Search Unit (SU) in U4 Format = U4, (e.g. 0x3 + 0xE = 0x1)

DWord	Bit	Description
	3:0	Search Path Distance [0] (X) – specifies relative X distance of the next walk from the starting position in unit of SU. Format = U4
1:13		Search Path Location [4 – 55] (X, Y)
14:31		**RD LUT SET 0-4**
14	31:24	LUT_MbMode [9] for Set 1 Format = U4U4 (encoded value must fit in 12-bits)
	23:16	LUT_MbMode [8] for Set 1 Format = U4U4 (encoded value must fit in 12-bits)
	15:8	LUT_MbMode [9] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)
	7:0	LUT_MbMode [8] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)
15	31:24	LUT_MbMode [9] for Set 3 Format = U4U4 (encoded value must fit in 12-bits)
	23:16	LUT_MbMode [8] for Set 3 Format = U4U4 (encoded value must fit in 12-bits)
	15:8	LUT_MbMode [9] for Set 2 Format = U4U4 (encoded value must fit in 12-bits)
	7:0	LUT_MbMode [8] for Set 2 Format = U4U4 (encoded value must fit in 12-bits)
16	31:24	LUT_MbMode [3] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)
	23:16	LUT_MbMode [2] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)

DWord	Bit	Description
	15:8	LUT_MbMode [1] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)
	7:0	LUT_MbMode [0] for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
17	31:24	LUT_MbMode [7] for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	23:16	LUT_MbMode [6] for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	15:8	LUT_MbMode [5] for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	7:0	LUT_MbMode [4] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)
18	31:24	LUT_MV [3] – For MV = 4 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	23:16	LUT_MV [2] – For MV = 2 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	15:8	LUT_MV [1] – For MV = 1 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	7:0	LUT_MV [0] – For MV = 0 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
19	31:24	LUT_MV [7] – For MV = 64 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	23:16	LUT_MV [6] – For MV = 32 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)

DWord	Bit	Description
	15:8	LUT_MV [5] – For MV = 16 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
	7:0	LUT_MV [4] – For MV = 8 for Set 0 Format = U4U4 (encoded value must fit in 10-bits)
20-23		**Finish RD LUT SET 1**
24-27		**Finish RD LUT SET 2**
28-31		**Finish RD LUT SET 3**

The assignment of LUT_MbMode entries is according to the MbTypeEx definition:

Index to LUT_MbMode	MbTypeEx	Description	AVC	VC1	MPEG2
0	MODE_INTRA_NONPRED	For INTRA8x8 and INTRA4x4 only. Added per 8x8 for INTRA8x8, and per 4x4 for INTRA4x4	Yes	n/a	n/a
1	MODE_INTRA MODE_INTRA_16x16	Added per 16x16 macroblock	Yes	Yes	Yes
2	MODE_INTRA_8x8	Added per 16x16 macroblock	Yes	n/a	n/a
3	MODE_INTRA_4x4	Added per 16x16 macroblock	Yes	n/a	n/a
8	MODE_INTER MODE_INTER_16x16	Added per 16x16 macroblock	Yes	Yes	Yes
9	MODE_INTER_BWD	Added for Refldx (per partition for major type or 8x8 for minor types)	Yes	Yes	Yes
4	MODE_INTER_16x8 MODE_INTER_8x16	Added per 16x16 macroblock	Yes	n/a	n/a
5	MODE_INTER_8x8q	Added per 8x8 subblock	Yes	Yes	n/a
6	MODE_INTER_8x4q	Added per 8x8 subblock	Yes	n/a	n/a
6	MODE_INTER_4x8q	Added per 8x8 subblock	Yes	n/a	n/a
7	MODE_INTER_4x4q	Added per 8x8 subblock	Yes	n/a	n/a
6	MODE_INTER_FIELD_16x8	Added per 16x16 macroblock	n/a	?	Yes
7	MODE_INTER_FIELD_8x8q	Added per 16x16 macroblock	n/a	n/a	n/a

The value of each byte of the LUTs will be viewed as a pair of 4-bit units: (shift, base), and constructed as base « shift.

For example, an entry 0x4A represents the value (0xA«0x4) = 10*16 = 160. Encoded value must fit in **12**-bits (unsigned number); otherwise, the hardware behavior is undefined.

The only exception is for Index of 9, MODE_INTER_BWD, which is used as a bias for the two search directions. It is a signed number instead, in the form of (SU3U4) = (sign, shift, base). The sign bit indicates whether the bias is added to the forward (if sign = 1) or the backward (if sign = 0). The bias has a magnitude of (base « shift), which has 11-bits precision. It should be noted that the number is always added, there is no subtraction.

Intra Modes only apply to AVC standard. The mode penalty doesn't apply to Skip Mode Checking. Note that while other mode penalty applies to a fixed macroblock partition, MODE_INTRA_NONPRED applies to all three intra modes. It is a constant cost adder for intra-mode coding regardless of the block size.

For source block that is less than 16x16 (like a 16x8 source block), the proper mode penalty that is stated as "added per 16x16 macroblock" is added once to the source block (like MODE_INTER_16x8 is added once to a 16x8 source block). It will not be divided by the source block size.

The LUT_MV is added to all motion vector coordinate deltas in quarter-pel unit except for the SKIP mode, which no costing penalty applies. Given motion vector coordinate, e.g. *mvx*, which is in quarter-pel precision (S5.2), the mv delta is defined to be its difference from the given costing center, e.g. *ccx*, and the costing penalty is applied to $dx = |mvx-ccx|$. The cost penalty is a piecewise linear interpolation from the LUT_MV table whereas the values on power-of-2 integer samples are provided. The piecewise linear interpolation is performed using quarter-pel precision, while the LUT_MV are only provided for the given power-of-2 integer positions. The maximum distance provided in the table is 64 pixels. A linear ramp with gradient of 1 on integer distance is applied for bigger distances with maximum penalty capped to 0x3FF (10 bits). Thus,

Costing_penalty_x = LUT_MV[int(dx)], if $dx < 3$ and dx = int(dx);

Costing_penalty_x = LUT_MV[p+1], else if $dx = 2^p$, for any p£6;.

Costing_penalty_x = LUT_MV[p+1] + ((LUT_MV[p+2] – LUT_MV[p+1])*k)»p,

else if $dx = 2^p+k$, for any $p<6$ and $k<.2^p$, and

Costing_penalty_x = min (LUT_MV[7] + int(dx)– 64, 255), else if $dx > 64$.

The total costing penalty for a motion vector is

Costing_penalty = Costing_penalty_x + Costing_penalty_y

As a convention, a (0,0) relative search path distance (meaning a repeat search path location) is treated as the ending of the search path. Or the search path may also end when **Max Predetermined Search Path Length** is reached, or one of the Early Success conditions is reached**.**

Software must program the search path to terminate with at least one (0,0).

Glossary of Messages

This section describes the glossary of messages in regard to Media Sampler.

Programming Note	
Context:	Glossary of Messages
Use of any messages to the Video Motion Estimation function while there are any messages to any sampler function is not allowed.	

Universal Input Message Phases

Major changes from the previous generation:

- Many fields are only required for one or two of the message types.
- MV cost and mode cost are moved into the message payload.
- RefID per block are new inputs.
- Enables for forward transform skip check, chroma searching.
- Thresholds and control data for forward transform skip check.
- Many of the performance thresholds have been removed (IME success, skip success, etc).

ValidMsgType = "..." identifies the given field is required for each message type. Hardware ignores these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

DWord	Bits	Description
M0.5	31:24	Reference Region Height (RefHeight): This field specifies the reference region height in pixels. When bidirectional search is enabled, this applies to both search regions. Minus 16 provides the number of search point in vertical direction. The value must be a multiple of 4. ValidMsgType = IME Format = U8 Range = [20, 64]
	23:16	Reference Region Width (RefWidth): This field specifies the search region width in pixels. When bidirectional search is enabled, this applies to both search regions. Minus 16 provides the number of search point in horizontal direction. The value must be a multiple of 4. ValidMsgType = IME Format = U8 Range = [20, 64] Note: Please make sure the reference windows are not completely outside of the video frame. In

DWord	Bits	Description
		that case, VME behavior is undefined. Note: Reference Window size must be <= Surface Size, otherwise VME behavior is undefined.
	15:8	Ignored
	7:0	Dispatch ID. This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. ValidMsgType = SIC, IME, FBR
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31	**2D 7 by 7 Enable** 0: Disable 1: Enable Notes: • Output phase W0 and W6 MBZ when this bit is asserted. • 2dsobel applies only to single reference and single start. • 2dsobel applies only to ref48x40.
	30:24	Sub-Macroblock Sub-Partition Mask (SubMbPartMask): This field defines the bit-mask for disabling sub-partition and sub-macroblock modes. The lower 4 bits are for the major partitions (sub-macroblock) and the higher 3 bits for minor partitions (with sub-partition for 4x(8x8) sub-macroblocks. xxxxxx1 : 16x16 sub-macroblock disabled xxxxx1x : 2x(16x8) sub-macroblock within 16x16 disabled xxxx1xx : 2x(8x16) sub-macroblock within 16x16 disabled xxx1xxx : 1x(8x8) sub-partition for 4x(8x8) within 16x16 disabled xx1xxxx : 2x(8x4) sub-partition for 4x(8x8) within 16x16 disabled x1xxxxx : 2x(4x8) sub-partition for 4x(8x8) within 16x16 disabled 1xxxxxx : 4x(4x4) sub-partition for 4x(8x8) within 16x16 disabled 1111111: Invalid Note: Invalid to have all partions disabled in the IME call. ValidMsgType = IME Usage Note: One example usage of only enabling 4x(4x4) sub-partition while all other partitions are disabled is for video processing, where parallel motion searches are performed for 16 4x4 blocks. For that no further block combination (into larger sub-partitions/sub-macroblocks) is needed.

DWord	Bits	Description
	23:22	Intra SAD Measure Adjustment (IntraSAD): This field specifies distortion measure adjustments used for the motion search SAD comparison. This field applies to both luma and chroma intra measurement. 00b: None 01b: Reserved 10b: Haar transform adjusted 11b: Reserved ValidMsgType = SIC
	21:20	Inter SAD Measure Adjustment (InterSAD):This field specifies distortion measure adjustments used for the motion search SAD comparison. This field applies to both luma and chroma intra measurement. 00b: None 01b: Reserved 10b: Haar transform adjusted 11b: Reserved ValidMsgType = SIC, IME, FBR, IDM Note: IDM msgs cannot have InterSAD set to 10b (Haar transform adjusted) if IdmSrcPixelMask is used. InterSAD must be set to 00b (None) if either IDMShapeMode5x5 or IDMShapeMode7x7 is enabled.
	19	Block-Based Skip Enabled: When this field is set on the skip thresholding passing criterion will be based on the maximal distortion of individual blocks (8x8's or 4x4's) instead of their sum (i.e. the distortion of 16x16). The block size is 8x8 if and only if the Transform 8x8 Flag is set to ON and the source size is 16x16. ValidMsgType = SIC
	18	BME disable for FBR Message (BMEDisableFBR): FBR messages that do not want bidirectional motion estimation performed will set this bit and VME will only perform fractional refinement on the shapes identified by subpredmode. Note: only the LSB of the subpredmode for each shape will be considred in FBR (a shape is either FWD or BWD as input of FBR, output however could change to BI if BME is enabled). 0 = BME enabled 1 = BME disabled ValidMsgType = FBR

DWord	Bits	Description
	17	Forward Transform Skip Check Enable (FTEnable): This field enables the forward transform calculation for skip check. It does not override the other skip calculations but it does decrease the performance marginially so don't enable it unless the transform is necessary. 0 = FT disabled 1 = FT enabled ValidMsgType = SIC
	16	**Process Inter Chroma Pixels Mode** (InterChroaZmaMode): This bit switches the inter operations from luma mode to chroma mode. All shapes sizes are referred to as UV pairs. For instance, the 4x4 shape is a 8x4 of pixel components (16 U and 16 V, interleaved vertically) and the 8x8 shape is a 16x8 of pixel components. MBMode is always 8x8. MBSubShape is either 8x8 or 4x4 indicated by LSB[1:0]. Bits[7:2] are MBZ. For MBSubShape of 4x4, SubPredMode is mapped to each 4x4 shape. Only 8x8 and 4x4 ModeCost are valid. Source block size is ignored. Streamin/streamout distortions are overloaded on 16x16 (Chroma8x8) and 8x8 (Chroma4x4). BilinearEnable is ignored (Chroma can only perform bilinear filtering) Restrictions when set: Intra operations are disabled (SIC), valid ref window sizes are 32x20, 24x24 (max Xsus), 16x32 (max Xsus), and 10x20 (max Xsus) (IME), adaptive is disabled (IME), no backward penalty cost (ALL), and only 4x4 and 8x8 shapes are valid (ALL). ValidMsgType = SIC IME, FBR
	15	Disable Field Cache Allocation: This field, when set to 1, disables the optimized field cache line method in the Sampler Cache for reference block data when RefAccess is 1 (field based). It is ignored by hardware if RefAccess is 0. 0 – Frame or field cache lines according to RefAccess 1 – Always frame cache lines ValidMsgType = IME, IDM
	14	**Skip Mode Type** For B_DIRECT_16x16, both motion vectors of the skip center pair 0 are used. For B_DIRECT_8x8s, all four skip center pairs are fully used (VME never tries to combine them with non-skip shapes from IME, FME, or BME).

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>0 : SKIP_1MVP – one MV pair for 16x16
1 : SKIP_4MVP – Four MV pairs for 8x8s (in this case and only this case, SkipCenter Delta 1-3 is used)
Note: SkipModeType should be programmed to 1MVP for non-16x16 Source size.
ValidMsgType = SIC</td></tr>
<tr><td></td><td>13:12</td><td>Sub-Pel Mode (SubPelMode)
This field defines the half/quarter pel modes. The mode is inclusive, ie., higher precision mode samples lower precision locations.
00b: Integer mode searching
01b: Half-pel mode searching
10b: Reserved
11b: Quarter-pel mode searching
ValidMsgType = FBR</td></tr>
<tr><td></td><td>11</td><td>Dual Search Path Option
Used only for dual record cases, this field flags whether two searching records uses the same or the different paths.
0: Use the same path as specified by the Search Path Location array
1: Use the different paths, the first one uses the even entries of the Search Path Location array and the second one uses the odd entries of the Search Path Location array.
ValidMsgType = IME</td></tr>
<tr><td></td><td>10:8</td><td>Search Control (SearchCtrl)
This field specifies how the motion search is performed.
ValidMsgType = IME
The following table shows the valid encodings. Other encodings are reserved.
<table>
<tr><th>Code</th><th>Mode</th></tr>
<tr><td>000b</td><td>Single reference, single record and single start.
Search is performed only on reference 0; only cost center 0 and start 0 are used. There is only one record. Adaptive search is also allowed. However, when AdaptiveEn is on, LenSU must be at least 2 as the adaptive search in VME is one-step delayed.
This is the common single directional motion search mode.</td></tr>
</table></td></tr>
</table>

DWord	Bits	Description
		001b: Single reference, single record and dual start. Search is performed only on reference 0; only cost center 0 is used. There is only one record. Search performs first on start 0 and then on start 1. Then if LenSP is not reached, the predetermined search path will start on start 1 with increment added to start 1 location. It then is followed by adaptive search. This is used for single direction adaptive search.
		011b: Single reference, dual record (and implied dual start). Search is performed only on reference 0; both cost center 0 and 1 and start 0 and 1 are used. Two records are used for both paths during IME. When integer search is complete, the two records are combined to find the best search. Sub-pel refinement is only performed from the best one. This may be used for search for multiple motion search candidates/predicators.
		111b: Dual reference, dual record (and implied dual start). Search is performed on references 0/1 with both cost centers 0/1 and starts 0/1. Two records are used for both paths during IME. When integer search is complete, and then sub-pel refinement is also performed separately, the two records are combined to find the best search on a subblock basis. This may be used for bidirectional motion search, or multi-reference P search. Whether bidirectional is enabled or not depends on the bidirection sub-macroblock mask. If BiSubMbPartMask is set to 1111'b, bidirectional search is disabled. VME outputs only the best unidirectional search results. Otherwise, BME is performed. Note that bidirectional search and sub-pel refinement are orthogonal features that can be enabled independently.
	7	**Reference Access (RefAccess)** This field defines how the reference blocks are accessed from the reference frames. It indicates if the source picture is a frame picture or a field picture. Programming Note: For all known video coding standards, reference pictures always have the same picture type as the source picture. Therefore, this field should be programmed to be the same as SrcAccess. 0: Frame based 1: Field based ValidMsgType = SIC, IME, FBR, IDM

DWord	Bits	Description
	6	**Source Access (SrcAccess)** This field defines how the source block is accessed from the source frame. It indicates if the source picture is a frame picture or a field picture. It is similar to the Picture Type used in video standards. 0: Frame based 1: Field based ValidMsgType = SIC, IME, FBR, IDM
	5:4	Inter MbType Remap (MbTypeRemap): This field controls the mapping of the output MbType when the VME output is an Inter (IntraMbFlag = INTER). The intended usage, for example, is for two forward (or backward) references or for two search regions from the same reference picture in one VME call. Hardware ignores this field if the VME output is an intra type (IntraMbFlag = INTRA). 00b: No remapping 01b: Remapping MbType to forward only (1-3 mapped to 1, even numbers in [4-14h] mapped to 4, odd numbers in [5-15h] mapped to 5, and 16h is unchanged) 10b: Remapping MbType to backward only (1-3 mapped to 2, even numbers in [4-14h] mapped to 6, odd numbers in [5-15h] mapped to 7, and 16h is unchanged) 11b: Reserved ValidMsgType = IME, FBR
	3	Reserved: MBZ
	2	Reserved: MBZ
	1:0	**Source Block Size (SrcSize)** This field defines how the 16x16 source block is formed. When Source Block Size is less than 16x16, SU larger than 4x4 is used. 00b: 16x16 01b: 16x8 10b: Reserved (for 8x16) 11b: 8x8 ValidMsgType = SIC, IME, FBR, IDM Note: For IDM message, the source block size should be always programmed to 16x16.

DWord	Bits	Description
M0.2	31:16	**Source Y (SrcY)** This field defines the vertical position (of the block's upper-left pixel) in units of pixels for the source block in the source frame. **Restriction** The Y address restriction is removed. Exception: for SIC messages where Intra Compute Type is set to 00 (Luma + Chroma enabled), SrcY must be a multiple of 2. ValidMsgType = SIC, IME, FBR, IDM Format = U16
	15:0	**Source X (SrcX)** This field defines the horizontal position (of the block's upper-left pixel) in units of pixels for the source block in the source picture. The source block must be within the source picture starting at any integer grid. For SIC messages where Intra Compute Type is set to 00 (Luma + Chroma enabled), SrcX must be a multiple of 2. ValidMsgType = SIC, IME, FBR, IDM Format = U16
M0.1	31:16	**Reference 1 Y Delta (Ref1Y)** This field defines the vertical position (of the upper-left corner of the reference region) in units of pixels for the Reference 1 region relative to the surface origin. **Restriction** The Y address restriction is removed. ValidMsgType = IME Format = S15 Hardware Range: [-2048 to 2047] Format = U16
	15:0	**Reference 1 X Delta (Ref1X)** This field defines the horizontal position (of the upper-left corner of the reference region) in units of pixels for the Reference 1 region relative to the surface origin. The resulting reference region is allowed to be outside the picture. Pixel replication is applied to

DWord	Bits	Description
		generate out of bound reference pixels. This field is only valid when dual reference mode is selected. Note: For search control=3, this must equal Ref0X. ValidMsgType = IME Format = S15 Hardware Range: [-2048 to 2047]
M0.0	31:16	**Reference 0 Y Delta (Ref0Y)** This field defines the vertical position (of the upper-left corner of the reference region) in units of pixels for the Reference 0 region relative to the surface origin. **Restriction** The Y address restriction is removed. ValidMsgType = IME, IDM Format = S15 Hardware Range: [-2048 to 2047] Format = U16
	15:0	**Reference 0 X Delta (Ref0X)** This field defines the horizontal position (of the upper-left corner of the reference region) in units of pixels for Reference 0 region relative to the surface origin. The resulting reference region is allowed to be outside the picture. Pixel replication is applied to generate out of bound reference pixels. ValidMsgType = IME, IDM Format = S15 Hardware Range: [-2048 to 2047]
M1.7	31:24	**Skip Center Enable Mask (SkipCenterMask):** [bits 31...24] xxxx xxx1: Ref0 Skip Center 0 is enabled [corresponds to M2.0] xxxx xx1x: Ref1 Skip Center 0 is enabled [corresponds to M2.1] xxxx x1xx: Ref0 Skip Center 1 is enabled [corresponds to M2.2] xxxx 1xxx: Ref1 Skip Center 1 is enabled [corresponds to M2.3] xxx1 xxxx: Ref0 Skip Center 2 is enabled [corresponds to M2.4]

DWord	Bits	Description
		xx1x xxxx: Ref1 Skip Center 2 is enabled [corresponds to M2.5] x1xx xxxx: Ref0 Skip Center 3 is enabled [corresponds to M2.6] 1xxx xxxx: Ref1 Skip Center 3 is enabled [corresponds to M2.7] Illegal cases: Disable both Ref0 and Ref1 Skip Center 0 in case of Skip_1MVP. Disable both Ref0 and Ref1 for any Skip Center pair in case of Skip_4MVP. ValidMsgType = SIC ValidMsgType = SIC
	23	IDM Shape Mode Select (IDMShapeMode): [Also see M1.1 bits 30 and 31] This bit selects what shape size the IDM is searching for. 0: 16x16 1: 8x8 ValidMsgType = IDM Note: Only ref window size of 32x32 (shape16x16), 24x24(shape8x8). Search control[2:0] defaults to single ref and single start. Luma only. 128x16 (shape16x16), and 32x16 (shape16x16) are supported. This bit must be set to 0 (i.e. shape 16x6) if it is IDM message type with ref window size of 128x16 and 32x16.
	22	**RefID Cost Mode Select (RefIDCostMode)** Selects the RefID costing mode. 0 = Mode0 (AVC) 1 = Mode1 (linear) ValidMsgType = SIC, IME, FBR, IDM
	21	**Enable AC-Only HAAR (AConlyHAAR)** This bit zeros out the DC component in the HAAR SATD block. 0 = AC+DC HAAR 1 = AC HAAR ValidMsgType = SIC, IME, FBR
	20	**Enable Weighted-SAD\HAAR (WeightedSADHAAR)** **Restriction** This bit enables weighted SAD\HAAR. 0: No weighted-SAD

DWord	Bits	Description
		1: Enable weighted-SAD Note: if this bit is 1, ShapeMask is ignored and only 16x16 shapes are accumulated (no partitioning). Restrictions: Only supported for source-type luma 16x16. Only support unidirectional search (on Ref0). See M1.3 for individual sub-block weight control. ValidMsgType = IME, IDM
	19	**Source Field Polarity Select (SrcFieldPolarity)** If SrcAccess = 1 (M0.3-6), meaning field based, than the hardware requires this value to derive the correct location on the source surface in memory to fetch pixels. This is because the source is stored as a frame picture with both fields interleaved in memory and the SrcY value (M0.2-31:16) is the location on the field picture (in other words, it does not convey the field polarity). Hence, the starting y-pixel coordinate fetched from memory is: SrcY* 2 + SrcFieldPolarity Else, this field is ignored by the hardware. ValidMsgType = SIC, IME, FBR, IDM Format = U1
	18	**Bilinear Filter Enable (BilinearEnable)** If set, the fractional filter implements a simple bilinear interpolation filter instead of the 4-tap filter. Note: This is supported for both hpel and qpel interpolation. ValidMsgType = SIC, FBR Format = Enable
	17:16	**MV Cost Scaling Factor (MVCostScaleFactor)** This term allows the user to redefine the precision of the lookup into the LUT_MV based on the MV cost difference from the cost center. The piecewise linear cost function is defined from 0 to 64 in powers of 2 intervals, and the precision of the difference is set by this field. There are 4 precision choices: 00b: Qpel [Qpel difference between MV and cost center: eff cost range 0-15pel] 01b: Hpel [Hpel difference between MV and cost center: eff cost range 0-31pel] 10b: Pel [Pel difference between MV and cost center: eff cost range 0-63pel] 11b: 2pel [2Pel difference between MV and cost center: eff cost range 0-127pel] ValidMsgType = SIC, IME, FBR, IDM

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>Format = U2</td></tr>
<tr><td></td><td>15:8</td><td>Macroblock Intra Structure (MbIntraStruct): This is a bitmask that specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.

Note: The user must set Bit6=Bit5.

The bit positions in the following table are relative positions within the field. For example, bit 7 in the table is bit 15 in the containing DWord.

<table><tr><th>Bit</th><th>MotionVerticalFieldSelect Index</th></tr><tr><td>7</td><td>Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)</td></tr><tr><td>6</td><td>Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)</td></tr><tr><td>5</td><td>IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)</td></tr><tr><td>4</td><td>IntraPredAvailFlagB – B (Upper neighbor)</td></tr><tr><td>3</td><td>IntraPredAvailFlagC – C (Upper left neighbor)</td></tr><tr><td>2</td><td>IntraPredAvailFlagD – D (Upper right neighbor)</td></tr><tr><td>1:0</td><td>Reserved: MBZ (ChromaIntraPredMode)</td></tr></table>
ValidMsgType = SIC</td></tr>
<tr><td></td><td>7</td><td>Luma Intra Source Corner Swap (IntraCornerSwap): This field specifies the format of the intra luma neighbor pixel format in the message.

0: Top neighbors are in sequential order.

1: Left-top corner is swapped with the last left-edge neighbor.

ValidMsgType = SIC</td></tr>
<tr><td></td><td>6</td><td>Non Skip MB Mode Cost Added (NonSkipModeAdded)

This field indicates that the distortion of the survived motion vectors becomes non-skip, and the MB mode cost is added to its distortion.

ValidMsgType = SIC</td></tr>
<tr><td></td><td>5</td><td>Non Skip Zero MV Cost Added (NonSkipZMvAdded)

This field indicates that the distortion of the survived motion vectors becomes non-skip, and the zero MV component costs are added to its distortion.

ValidMsgType = SIC</td></tr>
<tr><td></td><td>4:0</td><td>Luma Intra Partition Mask (IntraPartMask)

This field specifies which Luma Intra partition is enabled/disabled for intra mode decision.

xxxx1: luma_intra_16x16 disabled</td></tr>
</table>

DWord	Bits	Description
		xxx1x: luma_intra_8x8 disabled xx1xx: luma_intra_4x4 disabled Note: For SIC message with IntraComputeType == 00 or 01, at least 1 partition must be enabled. Bits [4:3] MBZ ValidMsgType = SIC
M1.6	31:28	**Bwd Block 3 RefID**
	27:24	**Fwd Block 3 RefID**
	23:20	**Bwd Block 2 RefID**
	19:16	**Fwd Block 2 RefID**
	15:12	**Bwd Block 1 RefID**
	11:8	**Fwd Block 1 RefID**
	7:4	**Bwd Block 0 RefID**
	3:0	**Fwd Block 0 RefID** M1.6 contains 8 input RefIDs, 1 per block. The RefID is used to penalize selection of shapes away from the optimal RefID similar to how MVCost penalizes shapes with motion vectors far from the cost center. **Restriction** Note: All 4 Bwd RefID are ignored by HW for IDM message type. Format = U4 ValidMsgType = SIC, IME, FBR, IDM
M1.5	31:16	Reserved: MBZ
	15:0	Reserved: MBZ
M1.4	31:16	Reserved: MBZ
	15:0	Reserved: MBZ
M1.3	31:30	**Weighted SAD Control Sub-block 15 (F)**
	29:28	**Weighted SAD Control Sub-block 14 (E)**
	27:26	**Weighted SAD Control Sub-block 13 (D)**

DWord	Bits	Description
	25:24	**Weighted SAD Control Sub-block 12 (C)**
	23:22	**Weighted SAD Control Sub-block 11 (B)**
	21:20	**Weighted SAD Control Sub-block 10 (A)**
	19:18	**Weighted SAD Control Sub-block 9**
	17:16	**Weighted SAD Control Sub-block 8**
	15:14	**Weighted SAD Control Sub-block 7**
	13:12	**Weighted SAD Control Sub-block 6**
	11:10	**Weighted SAD Control Sub-block 5**
	9:8	**Weighted SAD Control Sub-block 4**
	7:6	**Weighted SAD Control Sub-block 3**
	5:4	**Weighted SAD Control Sub-block 2**
	3:2	**Weighted SAD Control Sub-block 1**
	1:0	**Weighted SAD Control Sub-block 0** When the Weighted SAD control is enabled (M1.7 bit 20) these values are used to decrease the magnitude of each sub-block by dividing the 4x4 SAD\HAAR output mapped to that 4x4 of the source MB. The control value divides the 4x4 SAD\HAAR output by 2^control value. 0: »0 (div by 1) 1: »1 (div by 2) 2: »2 (div by 4) 3: »3 (div by 8) The output produces 1 16x16 macroblock results each with different weighted-SAD control. The values from M1.3 31:0 are mapped onto the sub-blocks of the source MB in the traditional Z-order: 16x16_0 Weighted-SAD Control Mapping: 0145 2367 89CD ABEF

DWord	Bits	Description
		ValidMsgType = IME, IDM
M1.2	31:28	**Start Center 1 Y (Start1Y)** This field defines the Y position of Search Path 1 relative to the reference Y location. It is in units of SU. ValidMsgType = IME Format = U4
	27:24	**Start Center 1 (Start1X)** This field defines the X position of Search Path 1 relative to the reference X location. It is in units of SU. The corresponding reference block must be fully within the reference region. ValidMsgType = IME Format = U4
	23:20	**Start Center 0 Y (Start0Y)** This field defines the Y position of Search Path 1 relative to the reference Y location. It is in units of SU. ValidMsgType = IME Format = U4
	19:16	**Start Center 0 X (Start0X)** This field defines the X position of Search Path 1 relative to the reference X location. It is in units of SU. The corresponding reference block must be fully within the reference region. ValidMsgType = IME Format = U4
	15:8	**Maximum Search Path Length (MaxNumSU)** This field defines the maximum number of SUs per reference including the predetermined SUs and the adaptively generated SUs. Note: Every SU in fixed path is counted (including the out-bound ones and repeated ones), and in addition for adaptive SUs only the ones actually searched are added. ValidMsgType = IME Format = U8, with valid range of [1,63]

DWord	Bits	Description
	7:0	**Max Fixed Search Path Length (LenSP)** This field defines the maximum number of SUs per reference which are evaluated by the predetermined SUs. When adaptive walk is enabled, adaptive walk starts when this number is reached. Note: Every SU in fixed path is counted (including the out-bound ones and repeated ones). ValidMsgType = IME Format = U8, with valid range of [1,63]
M1.1	31	**IDM Shape Mode 5x5 Select** (IDMShapeMode5x5) Extension This bit selects what shape size the IDM is searching for. For CenterWin, only 5x5 filter is supported. It is illegal to turn on both 5x5 and 7x7 filter at the same time. 0: Other 1: 5x5 ValidMsgType = IDM
	30	**IDM Shape Mode 7x7 Select (IDMShapeMode7x7) Extension** This bit selects what shape size the IDM is searching for. For SadofSobels, only 7x7 filter is supported. It is illegal to turn on both 5x5 and 7x7 filter at the same time. 0: Other 1: 7x7 ValidMsgType = IDM Note: The Source Block Size needs to be programmed to 16x8 if this is enabled.
	29	Reserved: MBZ
	28	Unidirectional Mix Disable (UniMixDisable): If it is on, all unidirectional resulting motion vectors must share the same direction, i.e. either all are forward, or all are backward. If this field is off, each partition, down to 8x8 subblock, may have a different mix of forward and backward motion vectors. (Within each 8x8 subblock, only one common choice is allowed.) Programming Note: For the case when BMEdisableFBR is set, only the input subpredmode direction is refined. If BMEdisableFBR is not set, both directions undergo fractional refinement before bidirectional refinement, but the subpredmode output never inverts directions if the refinement yields a better result (subpredmode could change to bidirectional in this case though). This field is MBZ except for cases of Search Control = 111'b (e.g. 7, dual reference). ValidMsgType = IME

DWord	Bits	Description
	27:24	RefPixelShift The amount the reference pixels are shifted positive or negative. The MSB implies right shift if zero or left shift if positive. The other three bits specify the amount that is shifted.
	23:22	Reserved: MBZ [Fixed7x7Weights -- GxMask, GyMask]
	21:16	**Bidirectional Weight (BiWeight)** This field defines the weighting for the backward and forward terms to generate the bidirectional term. This field is only valid for bidirectional search (SearchCtrl = 111). ValidMsgType = SIC, FBR Format = U6 Valid Values: [16, 21, 32, 43, 48]
	15:8	**RefId Polarity Bits** Bit15->bwd block3 Bit14->bwd block2 Bit13->bwd block1 Bit12->bwd block0 Bit11->fwd block3 Bit10->fwd block2 Bit9->fwd block1 Bit8->fwd block0 ValidMsg type - IME, SIC, FBR, IDM
	7	Reserved: MBZ
	6	Extended MV Cost Range This bit specifies if the increased 12-bit mvcost range is used vs. the legacy 10-bit range. 0 = Disable 1 = Enable ValidMsgType = SIC,IME, FBR
	5:0	**Maximum Number of Motion Vectors (MaxNumMVs)** This field specifies the maximum number of motion vectors allowed for the current macroblock. This field affects the macroblock partition decision. VME returns the best partition with MvQuantity

DWord	Bits	Description
		not exceeding MaxNumMVs. MaxNumMVs = 0 only allows skip as a valid Inter mode. Note: This value is used ONLY for 16x16 source MB mode. Usage Example: Certain profiles/levels for AVC standard have restrictions for the maximum number of motion vectors allowed for two consecutive macroblocks (MaxMvsPer2Mb may be 16 or 32). ValidMsgType = IME Format = U6
M1.0	31:24	**Early IME Successful Stop Threshold (EarlyImeStop)** This field specifies the threshold value for the IME distortion computes of single 16x16 mode below which no more search is performed within the IME unit. This field only takes effect if EarlyImeSuccessEn is set. Note: Early IME exit only looks at ref0, and uses 8x8 for source 8x8 and 2 16x8 0 for source 16x8. ValidMsgType = IME Format = U4U4 (encoded value should fit in 14 bits)
	23:16	Reserved: MBZ
	15:8	RefPixelOffset The amount the reference pixels are offset. This is a signed 8 bits so [-128,127].
	7	**Transform 8x8 Flag For Inter Enable (T8x8FlagForInterEn)** This field specifies whether Transform8x8Flag is updated for inter mode according to the resulting inter-mode sub-partition size. 0: Disable 1: Enable ValidMsgType = SIC, IME, FBR
	6	**X only search** This field enables searching in only the x dimension. ValidMsg type - IME, IDM X-only search supports only Lumma and non-adaptive search. This bit needs to be set to 0 for 1d-sobel and center-win.
	5	**Early IME Success Enable (EarlyImeSuccessEn)** This field specifies whether the Early Success may terminate on full-pel precision. When this field is not set, if early out does occur on full-pel location, hardware continues to local sub-pel refinement

DWord	Bits	Description
		search and so on. When this field is set, however, the local sub-pel refinement step is skipped and intra search is also skipped. This field only takes effect if EarlySuccessEn is set. Usage Example: This may be used for cases with large static area where (0,0) motion vector delivers very good results that no FME refinement is needed and also intra check is also skipped. This may also be used in place of Skip Mode Checking when the skip center(s) happens to be an integer location inside the SU of the Start Center(s). 0: Disable 1: Enable ValidMsgType = IME
	4:3	Reserved: MBZ
	2	Bidirectional Mix Disable (BiMixDis): If it is on, all resulting motion vectors must share the same direction, i.e. either all are unidirectional (i.e. forward or backward), or all bidirectional. If this field is off, each partition may have different search direction (forward, backward, or bidirectional). Usage Example: MPEG2 bidirectional decision is at whole macroblock level, while AVC decision is at subblock level. 0: Bidirectional decision on subblock level that bidirectional mode is enabled. 1: Bidirectional decision on whole macroblock. Note: This must be disabled for SubMbShape with any minors (8x4/4x8/4x4) in the MB. ValidMsgType = FBR
	1	Adaptive Search Enable (AdaptiveEn): This field defines whether adaptive searching is enabled for IME. When Adaptive Search is enabled, there must be at least two search steps preceded. It is either from a single start with step of >=2 or from a dual-start. 0: Disable 1: Enable ValidMsgType = IME
	0	Skip Mode Enable (SkipModeEn): This field specifies whether the skip mode checking is performed before the motion search. If this field is set, Skip Center, which may have a sub-pel precision, is first tested before IME. 0: Disable 1: Enable
M2.7	31:24	**SIC Forward Transform Coeff Threshold Matrix[6] - Highest Freq**

DWord	Bits	Description
	23:16	**SIC Forward Transform Coeff Threshold Matrix[5]**
	15:8	**SIC Forward Transform Coeff Threshold Matrix[4]**
	7:0	**SIC Forward Transform Coeff Threshold Matrix[3]**
M2.6	31:24	**SIC Forward Transform Coeff Threshold Matrix[2]**
	23:16	**SIC Forward Transform Coeff Threshold Matrix[1]**
	15:0	**SIC Forward Transform Coeff Threshold Matrix[0]** Values of the threshold matrix[0..6] are provided here. Matrix[0] contains the DC threashold for the Forward Transform Skip check. It has increased precision vs. the other thresholds due to the larger size of DC coeffieicients. Matrix[1] through Matrix[6] have lower precision. Threshold Matrix for 4x4 transform is as follows: 0 1 2 3 1 2 3 4 2 3 4 5 3 4 5 6 Matrix[0] Format = U16 Matrix[1..6] Format = U8 ValidMsgType = SIC
M2.5	31:24	Reserved: MBZ
	23:16	**FBR SubPredMode Input** VME uses this to select the appropriate shapes from the input message to perform FME on. Bits [1:0]: SubMbPredMode[0] Bits [3:2]: SubMbPredMode[1] Bits [5:4]: SubMbPredMode[2] Bits [7:6]: SubMbPredMode[3] 00: Forward 01: Backward 10: Bidirectional 11: Illegal

DWord	Bits	Description
		Note: Only the LSB of the subpredmode for each shape is considered in FBR (a shape is either FWD or BWD as input of FBR). ValidMsgType = FBR
	15:8	**FBR SubMBShape Input** This field is used to specify the subshape per block for fractional and bidirectional refinement. Bits [1:0]: SubMbShape[0] Bits [3:2]: SubMbShape[1] Bits [5:4]: SubMbShape[2] Bits [7:6]: SubMbShape[3] 00: 8x8 01: 8x4 10: 4x8 11: 4x4 ValidMsgType = FBR
	7:2	Reserved: MBZ
	1:0	**FBR MbMode Input** This field is used to specify the inter macroblock type in the same format as VME output. 00: 16x16 01: 16x8 10: 8x16 11: 8x8 ValidMsgType = FBR
M2.4	31:24	**MV 7 Cost**
	23:16	**MV 6 Cost**
	15:8	**MV 5 Cost**
	7:0	**MV 4 Cost**
M2.3	31:24	**MV 3 Cost**
	23:16	**MV 2 Cost**

DWord	Bits	Description
	15:8	**MV 1 Cost**
	7:0	**MV 0 Cost** Motion vector costings. See 6.3.3.1 for details. In short, the cost is linearly interpolated between control points. Format = U4U4 (encoded value must fit in 10 bits) **Note** ValidMsgType = SIC, IME, FBR, IDM
M2.2	31:24	**Chroma Intra Mode Cost** Penalty for chroma intra modes. DC = 0x Horz = 1x Vert = 1x Plane = 2x Format = U4U4 (encoded value must fit in 12-bits) ValidMsgType = SIC, IME, FBR
	23:16	**RefID Cost** RefID costing base penalty. Under AVC or Linear mode, different scaling are applied on top of this. Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR, IDM
	15:8	**Mode 9 Cost** MODE_INTER_BWD Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR
	7:0	**Mode 8 Cost** MODE_INTER_16x16 Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR, IDM
M2.1	31:24	**Mode 7 Cost** MODE_INTER_4x4q

DWord	Bits	Description
		MODE_INTER_FIELD_8x8q Format = U4U4 (encoded value must fit in 10 bits) ValidMsgType = SIC, IME, FBR
	23:16	**Mode 6 Cost** MODE_INTER_8x4q MODE_INTER_4x8q MODE_INTER_FIELD_16x8 Format = U4U4 (encoded value must fit in 10 bits) ValidMsgType = SIC, IME, FBR
	15:8	**Mode 5 Cost** MODE_INTER_8x8q Format = U4U4 (encoded value must fit in 10 bits) ValidMsgType = SIC, IME, FBR, IDM
	7:0	**Mode 4 Cost** MODE_INTER_16x8 MODE_INTER_8x16 Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR
M2.0	31:24	**Mode 3 Cost** MODE_INTRA_4x4 Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR
	23:16	**Mode 2 Cost** MODE_INTRA_8x8 Format = U4U4 (encoded value must fit in 12 bits) ValidMsgType = SIC, IME, FBR
	15:8	**Mode 1 Cost** MODE_INTRA_16x16 Format = U4U4 (encoded value must fit in 12 bits)

DWord	Bits	Description
		ValidMsgType = SIC, IME, FBR
	7:0	**Mode 0 Cost** MODE_INTRA_NONPRED Format = U4U4 (encoded value must fit in 10 bits) ValidMsgType = SIC, IME, FBR
M3.7	31:0	**BWD Cost Center 3**
M3.6	31:0	**FWD Cost Center 3**
M3.5	31:0	**BWD Cost Center 2**
M3.4	31:0	**FWD Cost Center 2**
M3.3	31:0	**BWD Cost Center 1**
M3.2	31:0	**FWD Cost Center 1**
M3.1	31:16	**BWD Cost Center 0 Delta Y (BWDCostCenter0Y)** This field defines the Y value for the first cost center relative to the picture source MB Y value for the BWD direction. All 4 Bwd Cost Center Deltas are ignored by HW for IDM message type. ValidMsgType = SIC, IME, FBR Format = S13.2 (2's comp) Hardware Range: [-512.00 to 511.75]
	15:0	**BWD Cost Center 0 Delta X (BWDCostCenter0X)** This field defines the X value for the first cost center relative to the picture source MB X value for the BWD direction. Major shape mapping to each cost center: CC0: 16x16_0, 16x8_0, 8x16_0, 8x8_0 CC1: 8x16_1, 8x8_1 CC2: 16x8_1, 8x8_2 CC3: 8x8_3

DWord	Bits	Description
		All 4 Bwd Cost Center Deltas are ignored by HW for IDM message type. ValidMsgType = SIC, IME, FBR Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75]
M3.0	31:16	FWDCostCenter 0 Delta Y (FWDCostCenter0Y): This field defines the Y value for the first cost center relative to the picture source MB Y value for the FWD direction. ValidMsgType = SIC, IME, FBR, IDM Format = S13.2 (2's comp) Hardware Range: [-512.00 to 511.75]
	15:0	FWDCostCenter 0 Delta X (FWDCostCenter0X): This field defines the X value for the first cost center relative to the picture source MB X value for the FWD direction. Major shape mapping to each cost center: CC0: 16x16_0, 16x8_0, 8x16_0, 8x8_0 CC1: 8x16_1, 8x8_1 CC2: 16x8_1, 8x8_2 CC3: 8x8_3 ValidMsgType = SIC, IME, FBR, IDM Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75]

SIC Input Message Phases

Major changes

- Addition of chroma pixel pairs (CbCr as 16b value) for the left 8, top 8, and top-left 1 corner.
- Addition of chroma mode masks (only 4 modes possible, so 4b mask).
- Addition of intra compute type (Y+CbCr, Y only, disabled).

ValidMsgType = "..." identifies the given field is required for each message type. Hardware will ignore these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

"X" in "WX+..." below is:

DWord	Bits	Name
WX+0.7	31:0	**Ref1 SkipCenter 3 Delta XY** Ref1 Skip Center 3 Delta Y: This field defines the Y value for the forward skip center relative to the 8x8 block offset from the source MB Y location in quarter-pel precision associated with Ref1. To match the relative 8x8 block location, the HW will add fixed offsets to the 4 skip centers in each direction to generate the correct pixel location to fetch the data. For SkipCenter 0: VME will add 0 to the user-input Y value. For SkipCenter 1: VME will add 0 to the user-input Y value. For SkipCenter 2: VME will add 32 to the user-input Y value. For SkipCenter 3: VME will add 32 to the user-input Y value. ValidMsgType = SIC Format = S13.2 (2's comp) Hardware Range: [-512.00 to 511.75] For chroma skip: Format = S12.3 (2's comp) Hardware Range: [-256.000 to 255.875] Ref1SkipCenter3 Delta X: This field defines the X value for the forward skip center relative to the 8x8 block offset from the source MB X location in quarter-pel precision associated with Ref1. To match the relative 8x8 block location, the HW will add fixed offsets to the 4 skip centers in each direction to generate the correct pixel location to fetch the data. For SkipCenter 0: VME will add 0 to the user-input X value. For SkipCenter 1: VME will add 32 to the user-input X value. For SkipCenter 2: VME will add 0 to the user-input X value. For SkipCenter 3: VME will add 32 to the user-input X value. Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75] For chroma skip: Format = S12.3 (2's comp) Hardware Range: [-1024.000 to 1023.875]
WX+0.6	31:0	**Ref0 SkipCenter 3 Delta XY***(for definition see M3.7)*

DWord	Bits	Name
WX+0.5	31:0	**Ref1 SkipCenter 2 Delta XY** *(for definition see M3.7)*
WX+0.4	31:0	**Ref0 SkipCenter 2 Delta XY** *(for definition see M3.7)*
WX+0.3	31:0	**Ref1 SkipCenter 1 Delta XY** *(for definition see M3.7)*
WX+0.2	31:0	**Ref0 SkipCenter 1 Delta XY** *(for definition see M3.7)*
WX+0.1	31:0	**Ref1 SkipCenter 0 Delta XY** *(for definition see M3.7)*
WX+0.0	31:0	**Ref0 SkipCenter 0 Delta XY** *(for definition see M3.7)*
WX+1.7	31:0	**Neighbor pixel Luma value [23, -1] to [20, -1].** Upper-right pixels from neighbor macroblock C
WX+1.6	31:0	**Neighbor pixel Luma value [19, -1] to [16, -1].** Upper-right edge pixels from neighbor macroblock C
WX+1.5	31:0	**Neighbor pixel Luma value [15, -1] to [12, -1].** Top edge pixels from neighbor macroblock B
WX+1.4	31:0	**Neighbor pixel Luma value [11, -1] to [8, -1].** Top edge pixels from neighbor macroblock B
WX+1.3	31:0	**Neighbor pixel Luma value [7, -1] to [4, -1].** Top edge pixels from neighbor macroblock B
WX+1.2	31:24	**Neighbor pixel Luma value [3, -1].** Fourth top edge pixel from neighbor macroblock B
	23:16	**Neighbor pixel Luma value [2, -1].** Third top edge pixel from neighbor macroblock B
	15:8	**Neighbor pixel Luma value [1, -1].** Second top edge pixel from neighbor macroblock B
	7:0	**Neighbor pixel Luma value [0, -1].** First top edge pixel from neighbor macroblock B
WX+1.1	31:24	**Corner Neighbor pixel 0.** Its content depends on IntraCornerSwap field. It swaps with Corner Neighbor pixel 1.
	23:10	Reserved: MBZ
	9:8	**Intra Compute Type (IntraComputeType)** This field specifies the pixel components measured for intra prediction. 00: Luma + Chroma enabled 01: Luma only

DWord	Bits	Name
		1X: Intra disabled
	7:4	**AVC Intra Chroma Mode Mask (IntraChromaModeMask)** The following mask disables the chroma intra modes from the output. xxx1: VERT xx1x: HORZ x1xx: DC 1xxx: PLANAR
	3:0	**AVC Intra 16x16 Mode Mask (Intra16x16ModeMask):** Disables given intra mode as follows. xxx1: xx1x: x1xx: 1xxx:
WX+1.0	31:25	Reserved: MBZ
	24:16	**AVC Intra 8x8 Mode Mask (Intra16x16ModeMask):** Disables given intra mode as follows. x xxxx xxx1: x xxxx xx1x: x xxxx x1xx: x xxxx 1xxx: x xxx1 xxxx: x xx1x xxxx: x x1xx xxxx: x 1xxx xxxx: 1 xxxx xxxx:
	15:9	Reserved: MBZ
	8:0	**AVC Intra 4x4 Mode Mask (Intra16x16ModeMask):** Disables given intra mode as follows.

DWord	Bits	Name
		x xxxx xxx1: x xxxx xx1x: x xxxx x1xx: x xxxx 1xxx: x xxx1 xxxx: x xx1x xxxx: x x1xx xxxx: x 1xxx xxxx: 1 xxxx xxxx:
WX+2.7	31:24	Reserved: MBZ
	23:16	Penalty for Intra4x4 non-DC prediction mode Format: U8
	15:8	Penalty for Intra8x8 non-DC prediction mode Format: U8
	7:0	Penalty for Intra16x16 non-DC prediction mode Format: U8
WX+2.6	31:0	Reserved: MBZ
WX+2.5	31:16	Reserved: MBZ
	15:0	**Neighbor pixel Chroma value CbCr pair [-1, -1]** Corner neighbor pixel pair (CbCr pair, each U8).
WX+2.4	31:28	**Intra Predictor Mode for Neighbor B15 (IntraMxMPredModeB15):** This field carries the intra prediction mode of the fourth bottom 4x4 block (Block 15 in Numbers of Block4x4 in a 16x16 region) of the top neighbor macroblock B. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification.
	27:24	**Intra Predictor Mode for Neighbor B14 (IntraMxMPredModeB14):** This field carries the intra prediction mode of the third bottom 4x4 block (Block 14 in Numbers of Block4x4 in a 16x16 region) of the top neighbor macroblock B. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification.
	23:20	**Intra Predictor Mode for Neighbor B11 (IntraMxMPredModeB11):** This field carries the intra prediction mode of the second bottom 4x4 block (Block 11 in Numbers of Block4x4 in a 16x16 region) of the top neighbor macroblock B. Definition of the term is according to Sections 8.3.1 and

DWord	Bits	Name
		8.3.2 of the AVC specification.
	19:16	**Intra Predictor Mode for Neighbor B10 (IntraMxMPredModeB10):** This field carries the intra prediction mode of the first bottom 4x4 block (Block 10 in Numbers of Block4x4 in a 16x16 region)of the top neighbor macroblock B. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification.
	15:12	**Intra Predictor Mode for Neighbor A15 (IntraMxMPredModeA15):** This field carries the intra prediction mode of the fourth rightmost 4x4 block (Block 15 in Numbers of Block4x4 in a 16x16 region) of the left neighbor A. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification.
	11:8	**Intra Predictor Mode for Neighbor A13 (IntraMxMPredModeA13):** This field carries the intra prediction mode of the third rightmost 4x4 block (Block 13 in Numbers of Block4x4 in a 16x16 region) of the left neighbor A. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification.
	7:4	**Intra Predictor Mode for Neighbor A7 (IntraMxMPredModeA7):** This field carries the intra prediction mode of the second rightmost 4x4 block (Block 7 in Numbers of Block4x4 in a 16x16 region) of the left neighbor A.
	3:0	**Intra Predictor Mode for Neighbor A5 (IntraMxMPredModeA5):** This field carries the intra prediction mode of the first rightmost 4x4 block (Block 5 in Numbers of Block4x4 in a 16x16 region) of the left neighbor A. Definition of the term is according to Sections 8.3.1 and 8.3.2 of the AVC specification. Intra Predictor Modes for Neighbor A and B are only used if **MODE_INTRA_NOPRED** is not zero. For intra mode selection, bias is applied to the predicted mode if a predictor is present for a partition. This is achieved by applying a penalty term MODE_INTRA_NONPRED defined in the VME state to the cost functions for non-predicted modes. The predictor for a given partition is from its left neighbor and top neighbor. The intra decision for a partition serves as the predictor for the next partition in the partition order as defined in Numbers of Block4x4 in a 16x16 region and Numbers of Block4x4 in an 8x8 region or numbers of Block8x8 in a 16x16 region. This set of intra predictor mode for neighbor macroblocks are only used for INTRA8x8 and INTRA4x4 modes. Format : U4 (The value of this field is defined in Definition of Intra4x4PredMode which is the same as that in Definition of Intra8x8PredMode.)
WX+2.3	31:24	**Corner Neighbor pixel 1.** Its content depends on **IntraCornerSwap** field. It swaps with **Corner Neighbor pixel 0**. **Neighbor pixel Luma value [-1, -1]**. The one upper-left edge pixel from neighbor macroblock D, which is the right most edge pixel of D, if **IntraCornerSwap** field is **1**. Or

DWord	Bits	Name
		Neighbor pixel Luma value [-1, 15]. The last left edge pixel from neighbor macroblock A, which is the left most edge pixel of D, if **IntraCornerSwap** field is 0.
	23:0	**Neighbor pixel Luma value [-1, 14] to [-1, 12].** Left edge pixels from neighbor macroblock A
WX+2.2	31:0	**Neighbor pixel Luma value [-1, 11] to [-1, 8].** Left edge pixels from neighbor macroblock A
WX+2.1	31:0	**Neighbor pixel Luma value [-1, 7] to [-1, 4].** Left edge pixels from neighbor macroblock A
WX+2.0	31:24	**Neighbor pixel Luma value [-1, 3].** Fourth left edge pixel from neighbor macroblock A
	23:16	**Neighbor pixel Luma value [-1, 2].** Third left edge pixel from neighbor macroblock A
	15:8	**Neighbor pixel Luma value [-1, 1].** Second left edge pixel from neighbor macroblock A
	7:0	**Neighbor pixel Luma value [-1, 0].** First left edge pixel from neighbor macroblock A
WX+3.7	31:0	**Neighbor pixel Chroma value CbCr pair [7, -1] to [6, -1]**
WX+3.6	31:0	**Neighbor pixel Chroma value CbCr pair [5, -1] to [4, -1]**
WX+3.5	31:0	**Neighbor pixel Chroma value CbCr pair [3, -1] to [2, -1]**
WX+3.4	31:0	**Neighbor pixel Chroma value CbCr pair [1, -1] to [0, -1]**
WX+3.3	31:0	**Neighbor pixel Chroma value CbCr pair [-1, 7] to [-1, 6]**
WX+3.2	31:0	**Neighbor pixel Chroma value CbCr pair [-1, 5] to [-1, 4]**
WX+3.1	31:0	**Neighbor pixel Chroma value CbCr pair [-1, 3] to [-1, 2]**
WX+3.0	31:0	**Neighbor pixel Chroma value CbCr pair [-1, 1] to [-1, 0]**

IME Input Message Phases

Major changes:

- Addition of the search path, no longer accessed via LUT, will come in message payload.
- Streamin\streamout now contains the 9 major shape reference indices per direction.
- Distortion precisions increased to 16b.

ValidMsgType = "..." identifies the given field is required for each message type. Hardware will ignore these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

"X" in "WX+..." below is:

DWord	Bits	Name
WX+0.7	31:0	**IME Search Path Delta 28-31**
WX+0.6	31:0	**IME Search Path Delta 24-27**
WX+0.5	31:0	**IME Search Path Delta 20-23**
WX+0.4	31:0	**IME Search Path Delta 16-19**
WX+0.3	31:0	**IME Search Path Delta 12-15**
WX+0.2	31:0	**IME Search Path Delta 8-11**
WX+0.1	31:0	**IME Search Path Delta 4-7**
WX+0.0	31:0	**IME Search Path Delta 0-3** [7:4] (Y) – specifies relative Y distance to the next SU from previous SU in units of SU. [3:0] (X) – specifies relative X distance to the next SU from previous SU in units of SU. Format = U8
WX+1.7	31:0	Reserved MBZ
WX+1.6	31:0	Reserved MBZ
WX+1.5	31:0	**IME Search Path Delta 52-55**
WX+1.4	31:0	**IME Search Path Delta 48-51**
WX+1.3	31:0	**IME Search Path Delta 44-47**
WX+1.2	31:0	**IME Search Path Delta 40-43**
WX+1.1	31:0	**IME Search Path Delta 36-39**
WX+1.0	31:0	**IME Search Path Delta 32-35**
WX+2.7	31:0	Reserved MBZ
WX+2.6	31:28	Rec0 Shape 8x8_3 RefID
	27:24	Rec0 Shape 8x8_2 RefID

DWord	Bits	Name
	23:20	Rec0 Shape 8x8_1 RefID
	19:16	Rec0 Shape 8x8_0 RefID
	15:12	Rec0 Shape 8x16_1 RefID
	11:8	Rec0 Shape 8x16_0 RefID
	7:4	Rec0 Shape 16x8_1 RefID
	3:0	Rec0 Shape 16x8_0 RefID Format = U4
WX+2.5	31:16	Rec0 Shape 16x16 Y (relative to source MB)
	15:0	Rec0 Shape 16x16 X (relative to source MB)
WX+2.4	31:20	Reserved MBZ
	19:16	Rec0 Shape 16x16 RefID Format = U4
	15:0	Rec0 Shape 16x16 Distortion Format = U16
WX+2.3	31:16	Rec0 Shape 8x8_3 Distortion Format = U16 Hardware only uses 14 bits. Upper bits ignored (True for all 8x8_X Distortions).
	15:0	Rec0 Shape 8x8_2 Distortion Format = U16
WX+2.2	31:16	Rec0 Shape 8x8_1 Distortion Format = U16
	15:0	Rec0 Shape 8x8_0 Distortion Format = U16
WX+2.1	31:16	Rec0 Shape 8x16_1 Distortion Format = U16 Hardware only uses 15 bits. Upper bits ignored (True for all 8x16_X Distortions).
	15:0	Rec0 Shape 8x16_0 Distortion

DWord	Bits	Name
		Format = U16
WX+2.0	31:16	Rec0 Shape 16x8_1 Distortion Format = U16 Hardware only uses 15 bits. Upper bits ignored (True for all 16x8_X Distortions).
	15:0	Rec0 Shape 16x8_0 Distortion Format = U16
WX+3.7	31:16	Rec0 Shape 8x8_3 Y (relative to source MB)
	15:0	Rec0 Shape 8x8_3 X (relative to source MB)
WX+3.6	31:16	Rec0 Shape 8x8_2 Y (relative to source MB)
	15:0	Rec0 Shape 8x8_2 X (relative to source MB)
WX+3.5	31:16	Rec0 Shape 8x8_1 Y (relative to source MB)
	15:0	Rec0 Shape 8x8_1 X (relative to source MB)
WX+3.4	31:16	Rec0 Shape 8x8_0 Y (relative to source MB)
	15:0	Rec0 Shape 8x8_0 X (relative to source MB)
WX+3.3	31:16	Rec0 Shape 8x16_1 Y (relative to source MB)
	15:0	Rec0 Shape 8x16_1 X (relative to source MB)
WX+3.2	31:16	Rec0 Shape 8x16_0 Y (relative to source MB)
	15:0	Rec0 Shape 8x16_0 X (relative to source MB)
WX+3.1	31:16	Rec0 Shape 16x8_1 Y (relative to source MB)
	15:0	Rec0 Shape 16x8_1 X (relative to source MB)
WX+3.0	31:16	Rec0 Shape 16x8_0 Y (relative to source MB)
	15:0	Rec0 Shape 16x8_0 X (relative to source MB)
WX+4.7	31:0	Reserved MBZ
WX+4.6	31:28	Rec1 Shape 8x8_3 RefID
	27:24	Rec1 Shape 8x8_2 RefID
	23:20	Rec1 Shape 8x8_1 RefID
	19:16	Rec1 Shape 8x8_0 RefID
	15:12	Rec1 Shape 8x16_1 RefID
	11:8	Rec1 Shape 8x16_0 RefID
	7:4	Rec1 Shape 16x8_1 RefID
	3:0	Rec1 Shape 16x8_0 RefID Format = U4

DWord	Bits	Name
WX+4.5	31:16	Rec1 Shape 16x16 Y (relative to source MB)
	15:0	Rec1 Shape 16x16 X (relative to source MB)
WX+4.4	31:20	Reserved MBZ
	19:16	Rec1 Shape 16x16 RefID Format = U4
	15:0	Rec1 Shape 16x16 Distortion Format = U16
WX+4.3	31:16	Rec1 Shape 8x8_3 Distortion Format = U16 Hardware only uses 14 bits. Upper bits ignored (True for all 8x8_X Distortions).
	15:0	Rec1 Shape 8x8_2 Distortion Format = U16
WX+4.2	31:16	Rec1 Shape 8x8_1 Distortion Format = U16
	15:0	Rec1 Shape 8x8_0 Distortion Format = U16
WX+4.1	31:16	Rec1 Shape 8x16_1 Distortion Format = U16 Hardware only uses 15 bits. Upper bits ignored (True for all 8x16_X Distortions).
	15:0	Rec1 Shape 8x16_0 Distortion Format = U16
WX+4.0	31:16	Rec1 Shape 16x8_1 Distortion Format = U16 Hardware only uses 15 bits. Upper bits ignored (True for all 16x8_X Distortions).
	15:0	Rec1 Shape 16x8_0 Distortion Format = U16

DWord	Bits	Name
WX+5.7	31:16	Rec1 Shape 8x8_3 Y (relative to source MB)
	15:0	Rec1 Shape 8x8_3 X (relative to source MB)
WX+5.6	31:16	Rec1 Shape 8x8_2 Y (relative to source MB)
	15:0	Rec1 Shape 8x8_2 X (relative to source MB)
WX+5.5	31:16	Rec1 Shape 8x8_1 Y (relative to source MB)
	15:0	Rec1 Shape 8x8_1 X (relative to source MB)
WX+5.4	31:16	Rec1 Shape 8x8_0 Y (relative to source MB)
	15:0	Rec1 Shape 8x8_0 X (relative to source MB)
WX+5.3	31:16	Rec1 Shape 8x16_1 Y (relative to source MB)
	15:0	Rec1 Shape 8x16_1 X (relative to source MB)
WX+5.2	31:16	Rec1 Shape 8x16_0 Y (relative to source MB)
	15:0	Rec1 Shape 8x16_0 X (relative to source MB)
WX+5.1	31:16	Rec1 Shape 16x8_1 Y (relative to source MB)
	15:0	Rec1 Shape 16x8_1 X (relative to source MB)
WX+5.0	31:16	Rec1 Shape 16x8_0 Y (relative to source MB)
	15:0	Rec1 Shape 16x8_0 X (relative to source MB)

FBR Input Message Phases

Major changes:

- Consists of the 32 sub-block motion vectors following the same 32MV format as the rest of VME.

ValidMsgType = "..." identifies the given field is required for each message type. Hardware will ignore these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

"X" in "WX+..." below is:

DWord	Bits	Name
WX+0.7	31:0	**Ref1 Sub-block XY 3**
WX+0.6	31:0	**Ref0 Sub-block XY 3**
WX+0.5	31:0	**Ref1 Sub-block XY 2**
WX+0.4	31:0	**Ref0 Sub-block XY 2**
WX+0.3	31:0	**Ref1 Sub-block XY 1**
WX+0.2	31:0	**Ref0 Sub-block XY 1**

DWord	Bits	Name
WX+0.1	31:0	**Ref1 Sub-block XY 0**
WX+0.0	31:16	**Ref0 Sub-block Y 0** The y-coordinate of Motion Vector 0 for Reference 0, relative to source MB location. **Note:** All MVs must be replicated for each shape. (e.g. for luma 16x16 shape and chroma 8x8, all Sub-block MVs must be the same. For luma 8x8 shape and chroma 4x4, each 8x8 must have its respective Sub-block MVs be replicated). Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75]
	15:0	**Ref0 Sub-block X 0** The x-coordinate of Motion Vector 0 for Reference 0, relative to source MB location. **Note**: All MVs must be replicated for each shape. (e.g. for luma 16x16 shape and chroma 8x8, all Sub-block MVs must be the same. For luma 8x8 shape and chroma 4x4, each 8x8 must have its respective Sub-block MVs be replicated). Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75]
WX+1.7	31:0	**Ref1 Sub-block XY 7**
WX+1.6	31:0	**Ref0 Sub-block XY 7**
WX+1.5	31:0	**Ref1 Sub-block XY 6**
WX+1.4	31:0	**Ref0 Sub-block XY 6**
WX+1.3	31:0	**Ref1 Sub-block XY 5**
WX+1.2	31:0	**Ref0 Sub-block XY 5**
WX+1.1	31:0	**Ref1 Sub-block XY 4**
WX+1.0	31:0	**Ref0 Sub-block XY 4**
WX+2.7	31:0	**Ref1 Sub-block XY 11**
WX+2.6	31:0	**Ref0 Sub-block XY 11**
WX+2.5	31:0	**Ref1 Sub-block XY 10**

DWord	Bits	Name
WX+2.4	31:0	**Ref0 Sub-block XY 10**
WX+2.3	31:0	**Ref1 Sub-block XY 9**
WX+2.2	31:0	**Ref0 Sub-block XY 9**
WX+2.1	31:0	**Ref1 Sub-block XY 8**
WX+2.0	31:0	**Ref0 Sub-block XY 8**
WX+3.7	31:0	**Ref1 Sub-block XY 15**
WX+3.6	31:0	Ref0 Sub-block XY 15
WX+3.5	31:0	Ref1 Sub-block XY 14
WX+3.4	31:0	Ref0 Sub-block XY 14
WX+3.3	31:0	Ref1 Sub-block XY 13
WX+3.2	31:0	Ref0 Sub-block XY 13
WX+3.1	31:0	Ref1 Sub-block XY 12
WX+3.0	31:0	Ref0 Sub-block XY 12

IDM Input Message Phases

Major changes:

- Consists of the 256b source pixel mask.

ValidMsgType = "..." identifies the given field is required for each message type. Hardware will ignore these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

"X" in "WX+..." below is:

DWord	Bits	Name
MX+0.7	31:0	**Source MB Pixel Mask Row 14, 15**
MX+0.6	31:0	**Source MB Pixel Mask Row 12, 13**
MX+0.5	31:0	**Source MB Pixel Mask Row 10, 11**
MX+0.4	31:0	**Source MB Pixel Mask Row 8, 9**
MX+0.3	31:0	**Source MB Pixel Mask Row 6, 7**

DWord	Bits	Name
MX+0.2	31:0	**Source MB Pixel Mask Row 4, 5**
MX+0.1	31:0	**Source MB Pixel Mask Row 2, 3**
MX+0.0	31:16	**Source MB Pixel Mask Row 1**
	15:0	**SourceMB Pixel Mask Row 0** These fields disable a given pixel of the 16x16 source MB. They are arranged from left to right with bit 0 being the left-most pixel and bit 15 being the right-most pixel of the source MB. Format = disable

Return Data Message Phases

Major changes:

Major Change
Many of the fields are not valid output for all message types.
Addition of new message phase, which has the block reference IDs and forward transform skip check data.
Intra chroma distortion and best mode are reported.
All U14 distortion values are now U16.

ValidMsgType = "..." identifies the given field is required for each message type. Hardware ignores these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

DWord	Bits	Description
W0.7	31:16	**Total VME Stalled Clocks:** Counts the number of clocks VME is stalled or starved while processing this request, due to cache misses. Format: U16 ValidMsgType = SIC, IME, FBR
W0.7	15:0	**Total VME Compute Clocks:** Counts the number of clocks VME is processing this request, but not stalled or starved as a result of cache misses. Format: U16 ValidMsgType = SIC, IME, FBR
W0.6	31:26	**Alternate Search Path Length:** Counts the number of unique search units computed by VME for the alternate search path for dual reference or dual search path. If the search path would return to a previously processed SU, it would not be reprocessed and hence not recounted. The value of

DWord	Bits	Description
		[W0.1 15:8] is the overall total search units processed from both paths whereas this value is the contribution only from the second search path. **Note:** Whenever VME is in a mode that processes only a single search path, this field is 0x0. Format: U6, Range of 0-48 ValidMsgType = IME
W0.6	25	**MaxMV Occurred:** This bit is set if the MaxMV event prevented the lowest distortion solution is rejected due to lack of motion vectors. Format: U1 Valid only for Luma Source Size = 16x16. ValidMsgType = IME
W0.6	24	**EarlyIMEStop Occurred:** This bit is set if the EarlyIMEStop threshold is satisfied and IME discontinues searching. Format: U1 ValidMsgType = IME
W0.6	23:16	**Sub-Macroblock Prediction Mode (SubMbPredMode):** If **InterMbMode** is INTER8x8, this field describes the prediction mode of the sub-partitions in the four 8x8 sub-macroblock. It contains four subfields each with 2 bits, corresponding to the four 8x8 sub-macroblocks in sequential order. This field is derived from sub_mb_type for a BP_8x8 macroblock. This field is derived from **MbType** for a non-BP_8x8 inter macroblock, and carries redundant information as **MbType**. If **InterMbMode** is INTER16x16, INTER16x8 or INTER8x16, this field carries the prediction modes of the sub macroblock (one 16x16, two 16x8, or two 8x16). The unused bits are set to zero. Bits [1:0]: SubMbPredMode[0] Bits [3:2]: SubMbPredMode[1] Bits [5:4]: SubMbPredMode[2] Bits [7:6]: SubMbPredMode[3] ValidMsgType = SIC, IME, FBR
W0.6	15:8	**Sub-Macroblock Shape (SubMbShape):** This field describes the subdivision of the four 8x8 sub-macroblocks. It contains four subfields each with 2 bits, corresponding to the four 8x8 sub macroblocks in sequential order.

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>This field is derived from sub_mb_type for a BP_8x8 or equivalent macroblock.

This field is forced to 0 for a non-BP_8x8 inter macroblock, and effectively carries redundant information as MbType.

This field is only valid if InterMbMode is INTER8x8; otherwise, it is set to zero.

Bits [1:0]: SubMbShape[0]

Bits [3:2]: SubMbShape[1]

Bits [5:4]: SubMbShape[2]

Bits [7:6]: SubMbShape[3]

ValidMsgType = SIC, IME, FBR</td></tr>
<tr><td>W0.6</td><td>7:0</td><td>Macroblock Intra Structure (MbIntraStruct): This is a bitmask that specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.

This field is simply copied from the input message (to reduce software overhead of forming the output message to PAK).

<table>
<tr><th>Bits</th><th>MotionVerticalFieldSelect Index</th></tr>
<tr><td>7</td><td>Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)</td></tr>
<tr><td>6</td><td>Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)</td></tr>
<tr><td>5</td><td>IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)</td></tr>
<tr><td>4</td><td>IntraPredAvailFlagB – B (Upper neighbor)</td></tr>
<tr><td>3</td><td>IntraPredAvailFlagC – C (Upper left neighbor)</td></tr>
<tr><td>2</td><td>IntraPredAvailFlagD – D (Upper right neighbor)</td></tr>
<tr><td>1:0</td><td>ChromaIntraPredMode</td></tr>
</table>

Note: This 8b field is MBZ when IntraComputeType == 1X (when intra is disabled).

ValidMsgType = SIC</td></tr>
<tr><td>W0.5</td><td>31:16</td><td>LumaIntraPredModes[3]</td></tr>
<tr><td>W0.5</td><td>15:0</td><td>LumaIntraPredModes[2]</td></tr>
<tr><td>W0.4</td><td>31:16</td><td>LumaIntraPredModes[1]</td></tr>
</table>

DWord	Bits	Description
W0.4	15:0	**LumaIntraPredModes[0]** Specifies the Luma Intra Prediction mode for four 4x4 sub-block, four 8x8 block or one intra16x16 of a MB. 4-bit per 4x4 sub-block (Transform8x8Flag=0, Mbtype=0 and intraMbFlag=1) or 8x8 block (Transform8x8Flag=1, Mbtype=0, MbFlag=1), since there are 9 intra modes. 4-bit for intra16x16 MB (Transform8x8Flag=0, Mbtype=1 to 24 and intraMbFlag=1), but only the LSBit[1:0] is valid, since there are only 4 intra modes. **Note:** The LumaIntraPredModes are MBZ when IntraComputeType == 1X (when intra is disabled). ValidMsgType = SIC
W0.3	31:16	**BestChromaIntraDistortion** This field provides the ChromaIntraMode distortion (sum of Cb and Cr dist). Note: this field is MBZ when IntraComputeType == 1X (when intra is disabled). Format = U16 ValidMsgType = SIC
W0.3	15:0	**BestIntraDistortion** This field provides redundant information. The IntraMbMode indicates if this is a 16x16/8x8/4x4 distortion. **Note:** This field is MBZ when IntraComputeType == 1X (when intra is disabled). Format = U16 ValidMsgType = SIC
W0.2	31:16	**SkipRawDistortion** This field contains Skip Raw Distortion which may be used by software to further refine the skip decision. **Note:** this field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U16 ValidMsgType = SIC
W0.2	15:0	**InterDistortion** This field provides the best inter distortion. Format = U16 ValidMsgType = SIC, IME, FBR
W0.1	31:27	Reserved: MBZ

DWord	Bits	Description
W0.1	26:16	**Sum Ref1 Inter Dist Upper 10 bits (SumInterDistL1Upper)** Contains the sum distortion of all 16x16 Inter shape of Ref1 within the searched SU of this search window. MSB 11 bits only. Format = U11 ValidMsgType = IME
W0.1	15:8	**Search Path Length:** This field returns the number of SU it takes in the integer search. It includes predetermined search path and dynamic search path. Format: U8 ValidMsgType = IME
W0.1	7:4	**Reference 1 border reached:** Bitmask indicating whether any border of reference 1 is reached by one or more motion vectors in the winning inter mode. xxx1: left border reached xx1x: right border reached x1xx: top border reached 1xxx: bottom border reached ValidMsgType = IME
W0.1	3:0	**Reference 0 border reached:** Bitmask indicating whether any border of reference 0 is reached by one or more motion vectors in the winning inter mode. xxx1: left border reached xx1x: right border reached x1xx: top border reached 1xxx: bottom border reached ValidMsgType = IME
W0.0	31	**StreamOutHasNoLocal:** Stream In is the stream out.
W0.0	30	**MinHasLocal:** Part of the results for the minimum distortion contains local results.
W0.0	29	**StreamOutOnlyLocal:** The stream out results don't have any of the stream in results included.
W0.0	28:24	**MvQuantity:** Specifies the number of MVs in packed format (in units of motion vectors). **Note:** This field is provided to help software meet conformance requirements such as maximum number of motion vectors for two consecutive macroblocks.

DWord	Bits	Description
		Format: U5, valid from 0 to 32 ValidMsgType = SIC, IME, FBR
W0.0	23	**ExtendedForm:** This field specifies that **LumaIntraMode**'s are fully replicated in 4x4 sub-blocks respectively. And motion vectors must be in unpacked form as well. This non-DXVA form is used for optimal kernel performance. **This is reserved MBZ and the HW always extends.** ValidMsgType = SIC
W0.0	22:21	Reserved: MBZ
W0.0	20:16	**IntraMbType** This field is encoded to match with the inter type determined as described in the next section. It follows a unified encoding for intra macroblocks according to the AVC Spec. **Note:** This field is MBZ when IntraComputeType == 1X (when intra is disabled). ValidMsgType = SIC
W0.0	15	**Transform8x8Flag (Transform 8x8 Flag)** This field indicates that 8x8 transform is recommended. It is set to 1 if **IntraMbFlag** = INTRA and **IntraMbMode** = INTRA_8x8. For **IntraMbFlag** = INTER. If **T8x8FlagForInterEn** = 0, this field is set to 0 by VME hardware. If **T8x8FlagForInterEn** = 1, this field is set to 1 if there is no sub macroblock size less than 8x8 (noSubMbPartSizeLessThan8x8Flag = 1). 0: 4x4 integer transform 1: 8x8 integer transform **Note:** This bit is always 0 for non-16x16 source block cases. ValidMsgType = IME, FBR
W0.0	14	**FieldMbFlag** This field indicates the inter prediction result is field or frame. It is always set to **SrcAccess**. 0: Frame macroblock 1: Field macroblock ValidMsgType = SIC, IME, FBR
W0.0	13	Reserved: MBZ
W0.0	12:8	**InterMbType**

DWord	Bits	Description
		This field is encoded to match with the inter type determined as described in the next section. It follows a unified encoding for inter macroblocks according to AVC Spec. ValidMsgType = SIC, IME, FBR
W0.0	7	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Unique for AVC standard, within an MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in an MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture in most coding standard, this field is a constant for the whole field picture. It is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. This field is reserved and MBZ for a progressive frame picture. VME hardware set this field to 1 if the source block is a field block from the bottom field and otherwise sets it to 0. This is accomplished by the following equation using input signals **SrcAccess** and **SrcY**: **SrcAccess** && (bit0(**SrcY**) ==1). 0 = Current macroblock is a field macroblock from the **top** field. 1 = Current macroblock is a field macroblock from the **bottom** field. Equals SrcAccess && SrcFieldPolarity(M1.7[19]) ValidMsgType = SIC, IME, FBR
W0.0	6	Reserved: MBZ
W0.0	5:4	**IntraMbMode** This field is provided to carry redundant information as that in **MbType**. The full extended definition of this field allows kernel software to help update the **MbType** field when outputting controls to the MFX PAK encoding. VME outputs this field regardless of MbIntraFlag value if intra mode is enabled. ValidMsgType = SIC
W0.0	3:2	Reserved: MBZ
W0.0	1:0	**InterMbMode** This field is provided to carry redundant information as that in **MbType**. The full extended definition of this field allows kernel software to help update the **MbType** field when outputting controls to the MFX PAK encoding. VME outputs this field regardless of MbIntraFlag value if inter mode is enabled. ValidMsgType = SIC, IME, FBR
W1.7 to W1.2	31:0 Each	**MVb[3]** to **MVb[1]**. Motion vectors 3 to 1 for Reference 1, and

DWord	Bits	Description
		MVa[3] to **MVa[1]**. Motion vectors 3 to 1 for Reference 0 See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W1.1	31:16	**MVb[0].y:** Returning the y-coordinate of Motion Vector 0 for Reference 1, relative to source MB location. Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75] ValidMsgType = SIC, IME, FBR
W1.1	15:0	**MVb[0].x:** Returning the x-coordinate of Motion Vector 0 (co-located w/ sublbock_4x4_0) for Reference 1, relative to source MB location. Its meaning is determined by **MbType**. Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75] ValidMsgType = SIC, IME, FBR
W1.0	31:16	**MVa[0].y:** Returning the y-coordinate of Motion Vector 0 for Reference 0, relative to source MB location. Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75] ValidMsgType = SIC, IME, FBR
W1.0	15:0	**MVa[0].x:** Returning the x-coordinate of Motion Vector 0 (co-located w/ the first pixel in 6 by 2 block) for Reference 0, relative to source MB location. Its meaning is determined by **MbType**. Hardware Range: [-2048.00 to 2047.75] ValidMsgType = IME **MVa[0].x:** Returning the x-coordinate of Motion Vector 0 (co-located w/ sublbock_4x4_0) for Reference 0, relative to source MB location. Its meaning is determined by **MbType**. The returned motion vectors are placed in a fixed data format, with up to 16 motion vectors for one reference and the motion vectors from reference 0 and 1 interleaved. If M1.7:20 (Enable WeightedSAD) is set true (1), then this field and the rest of W1 through W4 are redefined as follows: W1.0 from 31:0 = Minimum 16x16 distortion when applying 1st weighting pattern (16x16_0) W1.1 from 31:0 = Reserved

DWord	Bits	Description
		W1.2 from 31:0 = Minimum 16x16 distortion when applying 2nd weighting pattern (16x16_1) W1.3 from 31:0 = Reserved W1.4 from 31:0 = Minimum 16x16 distortion when applying 3rd weighting pattern (16x16_2) W1.5 from 31:0 = Reserved W1.6 from 31:0 = Minimum 16x16 distortion when applying 4th weighting pattern (16x16_3) W1.7 from 31:0 = Reserved W2 through W4 = Reserved If M0.3:31 (2D 7 by 7 enable) is set true (1), then this field and the rest of W1 through W4 are redefined as follows: W1.0 from 15:0 = returning the x-coordinate of Motion Vector 0 (co-located w/ the first pixel in 6 by 2 block) for Reference 0, relative to source MB location. W1.0 from 31:16 = returning the y-coordinate of Motion Vector 0 (co-located w/ the first pixel in 6 by 2 block) for Reference 0, relative to source MB location. W1.1 from 31:0 = Reserved W1.2 from 31:0 = MVa[1] W1.3 from 31:0 = Reserved W1.4 from 31:0 = MVa[2] W1.5 from 31:0 = Reserved W1.6 from 31:0 = MVa[3] W1.7 from 31:0 = Reserved W2.0 from 31:0 = MVa[4] W2.1 from 31:0 = Reserved W2.2 from 31:0 = MVa[5] W2.3 from 31:0 = Reserved W2.4 from 31:0 = MVa[6] W2.5 from 31:0 = Reserved W2.6 from 31:0 = MVa[7]

DWord	Bits	Description
		W2.7 from 31:0 = Reserved W3.0 from 31:0 = MVa[8] W3.1 from 31:0 = Reserved W3.2 from 31:0 = MVa[9] W3.3 from 31:0 = Reserved W3.4 from 31:0 = MVa[10] W3.5 from 31:0 = Reserved W3.6 from 31:0 = MVa[11] W3.7 from 31:0 = Reserved W4.0 to W4.7 bits 31:0 = Reserved Format = S13.2 (2's comp) Hardware Range: [-2048.00 to 2047.75] ValidMsgType = SIC, IME, FBR
W2.7 to W2.0	31:0 Each	**MVb[7]** to **MVb[4]**. Motion vectors 7 to 4 for Reference 1, and **MVa[7]** to **MVa[4]**. Motion vectors 7 to 4 for Reference 0 See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W3.7 to W3.0	31:0 Each	**MVb[11]** to **MVb[8]**. Motion vectors 11 to 8 for Reference 1, and **MVa[11]** to **MVa[8]**. Motion vectors 11 to 8 for Reference 0 See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W4.7 to W4.0	31:0 Each	**MVb[15]** to **MVb[12]**. Motion vectors 15 to 12 for Reference 1, and **MVa[15]** to **MVa[12]**. Motion vectors 15 to 12 for Reference 0 See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W5.7 to W5.1	31:0 Each	**InterDistortion[15] to InterDistortion[2]**. Inter-prediction-distortion associated with motion vector 15 to 2. Its meaning is determined by sub-shape.

DWord	Bits	Description
		See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W5.0	31:16	**InterDistortion[1].** Inter-prediction-distortion with motion vector 1 (co-located with subblock_4x4_1). Its meaning is determined by sub-shape. Format = U16 See note in W1.0 bits 15:0. ValidMsgType = SIC, IME, FBR
W5.0	15:0	**InterDistortion[0].** Inter-prediction-distortion associated with motion vector 0 (co-located with subblock_4x4_0). Its meaning is determined by sub-shape. It must be zero if the corresponding sub-shape is not chosen. This field may be associated with MVa[0] and/or MVb[0], depending on the resulting prediction mode for the sub-block. If the corresponding MV field is created by "duplication", this field must be zero. For 1MVP skip messages, the 16x16 distortion (sad + mv cost + ref cost) is present here. For 4MVP skip messages, the 4 8x8 distortions (sad + mv cost + ref cost) are present here. **Note:** This set of inter-prediction-distortion fields contains detailed information for all sub-shapes. It may be used to assist a multi-pass motion search algorithm. The overall distortion for the macroblock is provided in W0.1. **Format = U16** If M1.7:20 (Enable WeightedSAD) is set true (1), then this field and the rest of W5 are redefined as follows: W5.0 from 15:0 = Minimum 16x16 distortion when applying 1st weighting pattern (16x16_0 with corresponding MV at W1.0) W5.0 from 31:16 = Minimum 16x16 distortion when applying 2nd weighting pattern (16x16_1 with corresponding MV at W1.2) W5.1 from 15:0 = Minimum 16x16 distortion when applying 3rd weighting pattern (16x16_2 with corresponding MV at W1.4) W5.1 from 31:16 = Minimum 16x16 distortion when applying 4th weighting pattern (16x16_3 with corresponding MV at W1.6) W5.7 to W5.2 = Reserved If M0.3:31 (2D 7 by 7 enable) is set true (1), then this field and the rest of W5 is redefined as follows:

DWord	Bits	Description
		W5.0 from 15:0 = **InterDistortion[0]**- Inter-prediction-distortion associated with motion vector 0 (First pixel in 6 by 2 block). W5.0 from 31:16 = **InterDistortion[1]** -Inter-prediction-distortion with motion vector 1 (Second pixel in 6 by 2 block). Its meaning is determined by sub-shape. W5.1 to W5.5 = **InterDistortion[2] to InterDistortion[11]** W5.1 to W5.5 = Reserved ValidMsgType = SIC, IME, FBR
W6.7	31:16	**Max Ref1 Inter Dist (MaxRef1InterDist)** Contains the distortion of the 16x16 Inter shape of Ref1 with the maximum distortion within the searched SU of this search window. Format = U16 ValidMsgType = IME
W6.7	15:0	**Max Ref0 Inter Dist (MaxRef0InterDist)** Contains the distortion of the 16x16 Inter shape of Ref0 with the maximum distortion within the searched SU of this search window. Format = U16 ValidMsgType = IME
W6.6	31:27	Reserved: MBZ
W6.6	26:0	**Sum Ref0 Inter Dist (SumRef0InterDist)** Contains the sum distortion of all 16x16 Inter shape of Ref0 within the searched SU of this search window. Format = U27 ValidMsgType = IME
W6.5	31:16	**Block 0 Chroma Cr Coeff Magnitude Clip Sum** Sum of how much all the coefficients across 1 block exceeded their respective threshold. **Note:** This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U16 ValidMsgType = SIC
W6.5	15:0	**Block 0 Chroma Cb Coeff Magnitude Clip Sum** Sum of how much all the coefficients across 1 block exceeded their respective threshold.

DWord	Bits	Description
		Note: This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U16 ValidMsgType = SIC
W6.4	31:16	**Sum Ref1 Inter Dist lower 16 bits (SumInterDistL1Lower)** Contains the sum distortion of all 16x16 Inter shape of Ref1 within the searched SU of this search window. Lower 16 bits. Format = U16 ValidMsgType = IME
W6.4	15:8	**Block 0 Chroma Cr NZC** Count of the coefficients across 1 block that exceeded their respective threshold. **Note:** This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U8 ValidMsgType = SIC
W6.4	7:0	**Block 0 Chroma Cb NZC** Count of the coefficients across 1 block that exceeded their respective threshold. **Note:** This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U8 ValidMsgType = SIC
W6.3	31:16	**Block 3 Luma Coeff Magnitude Clip Sum**
W6.3	15:0	**Block 2 Luma Coeff Magnitude Clip Sum**
W6.2	31:16	**Block 1 Luma Coeff Magnitude Clip Sum**
W6.2	15:0	**Block 0 Luma Coeff Magnitude Clip Sum** Sum of how much all the coefficients across 1 block exceeded their respective threshold. **Note:** This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U16 ValidMsgType = SIC
W6.1	31:24	**Block 3 Luma NZC**

DWord	Bits	Description
W6.1	23:16	**Block 2 Luma NZC**
W6.1	15:8	**Block 1 Luma NZC**
W6.1	7:0	**Block 0 Luma NZC** Count of the coefficients across 1 block that exceeded their respective threshold. **Note:** This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U8 ValidMsgType = SIC
W6.0	31:28	**Bwd Block 3 RefID**
W6.0	27:24	**Fwd Block 3 RefID**
W6.0	23:20	**Bwd Block 2 RefID**
W6.0	19:16	**Fwd Block 2 RefID**
W6.0	15:12	**Bwd Block 1 RefID**
W6.0	11:8	**Fwd Block 1 RefID**
W6.0	7:4	**Bwd Block 0 RefID** Reference ID for backward block 0. **Note:** Even if shape is 16x16, this field is defined per block, hence VME will replicate the RefID for larger shapes. Replication happens only in IME. For CRE (SIC/FBR), this is a pass through field. Format = U4 ValidMsgType = SIC, IME, FBR
W6.0	3:0	**Fwd Block 0 RefID** Reference ID for forward block 0. **Note:** Even if shape is 16x16, this field is defined per block, hence VME will replicate the RefID for larger shapes. Replication happens only in IME. For CRE (SIC/FBR), this is a pass through field.

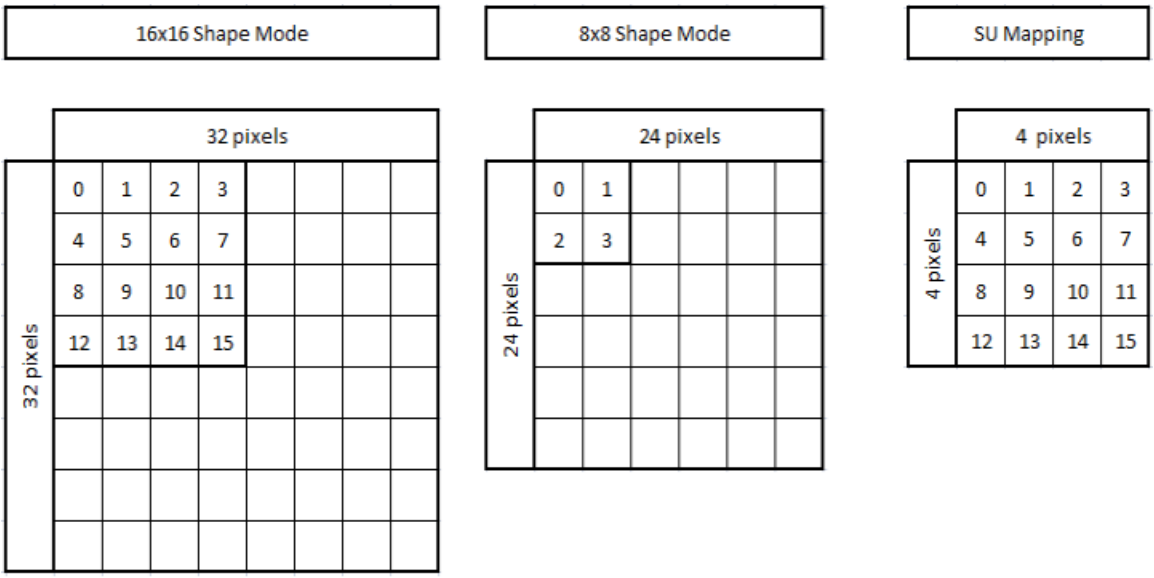
DWord	Bits	Description
		Format = U4 ValidMsgType = SIC, IME, FBR

IME StreamOut

IME Streamout follows the same format as the IME Streamin message phases (IME2-IME5).

IDM Stream-Out

Because the IDM output message only contains distortion values without their corresponding motion vectors, a direct mapping system is used to derive the MV associated with a given distortion.



This mapping allows the user to derive the location of a given distortion easily by applying the following equations.

ShapeMode == 16x16	ShapeMode == 8x8
MV.x = (GRF# % 4) * 4 + W# % 4 + Ref.x	**MV.x = ((GRF# / 4) % 2) * 4 + W# % 4 + Ref.x**
MV.y = (GRF# / 4) * 4 + W# / 4 + Ref.y	**MV.y = ((GRF# / 4) / 2) * 4 + W# / 4 + Ref.y**

- GRF#: The 256b return message phase number.
- W#: The 16 16b word within each 256b return message.

IDM16x16 Streamout Message Format

DWord	Bits	Description
W15	31:0	**Distortion Mesh Block 15 Search Points 0-15**
W14	31:0	**Distortion Mesh Block 14 Search Points 0-15**
W13	31:0	**Distortion Mesh Block 13 Search Points 0-15**
W12	31:0	**Distortion Mesh Block 12 Search Points 0-15**
W11	31:0	**Distortion Mesh Block 11 Search Points 0-15**
W10	31:0	**Distortion Mesh Block 10 Search Points 0-15**
W9	31:0	**Distortion Mesh Block 9 Search Points 0-15**
W8	31:0	**Distortion Mesh Block 8 Search Points 0-15**
W7	31:0	**Distortion Mesh Block 7 Search Points 0-15**
W6	31:0	**Distortion Mesh Block 6 Search Points 0-15**
W5	31:0	**Distortion Mesh Block 5 Search Points 0-15**
W4	31:0	**Distortion Mesh Block 4 Search Points 0-15**
W3	31:0	**Distortion Mesh Block 3 Search Points 0-15**
W2	31:0	**Distortion Mesh Block 2 Search Points 0-15**
W1	31:0	**Distortion Mesh Block 1 Search Points 0-15**
W0.7	31:16	**Distortion Mesh Block 0 Search Point 15**
	15:0	**Distortion Mesh Block 0 Search Point 14**
W0.6	31:16	**Distortion Mesh Block 0 Search Point 13**
	15:0	**Distortion Mesh Block 0 Search Point 12**
W0.5	31:16	**Distortion Mesh Block 0 Search Point 11**

DWord	Bits	Description
	15:0	**Distortion Mesh Block 0 Search Point 10**
W0.4	31:16	**Distortion Mesh Block 0 Search Point 9**
	15:0	**Distortion Mesh Block 0 Search Point 8**
W0.3	31:16	**Distortion Mesh Block 0 Search Point 7**
	15:0	**Distortion Mesh Block 0 Search Point 6**
W0.2	31:16	**Distortion Mesh Block 0 Search Point 5**
	15:0	**Distortion Mesh Block 0 Search Point 4**
W0.1	31:16	**Distortion Mesh Block 0 Search Point 3**
	15:0	**Distortion Mesh Block 0 Search Point 2**
W0.0	31:16	**Distortion Mesh Block 0 Search Point 1**
	15:0	**Distortion Mesh Block 0 Search Point 0** This is the distortion value at the location (0,0) within the search window. Each 256b (8 DW) message phase contains 16 distortion values for a given 4x4 search unit of the search window. Each word of a message phase contains a single 16x16 distortion at a given search point. The individual distortion values are arranged in raster-scan (row-major order) within the 4x4. For ShapeMode == 16, the 16 message phases comprise the 16 SUs (4x4) of the 32x32 search window for each 16x16 distortion and are returned in raster-scan (row-major order). Format = U16

IDM8x8 Streamout Message Format

DWord	Bits	Description
W15	31:0	**Distortion Mesh Block 3 SU3 Search Points 0-15**
W14	31:0	**Distortion Mesh Block 2 SU3 Search Points 0-15**
W13	31:0	**Distortion Mesh Block 1 SU3 Search Points 0-15**
W12	31:0	**Distortion Mesh Block 0 SU3 Search Points 0-15**
W11	31:0	**Distortion Mesh Block 3 SU2 Search Points 0-15**

DWord	Bits	Description
W10	31:0	**Distortion Mesh Block 2 SU2 Search Points 0-15**
W9	31:0	**Distortion Mesh Block 1 SU2 Search Points 0-15**
W8	31:0	**Distortion Mesh Block 0 SU2 Search Points 0-15**
W7	31:0	**Distortion Mesh Block 3 SU1 Search Points 0-15**
W6	31:0	**Distortion Mesh Block 2 SU1 Search Points 0-15**
W5	31:0	**Distortion Mesh Block 1 SU1 Search Points 0-15**
W4	31:0	**Distortion Mesh Block 0 SU1 Search Points 0-15**
W3	31:0	**Distortion Mesh Block 3 SU0 Search Points 0-15**
W2	31:0	**Distortion Mesh Block 2 SU0 Search Points 0-15**
W1	31:0	**Distortion Mesh Block 1 SU0 Search Points 0-15**
W0.7	31:16	**Distortion Mesh Block 0 SU0 Search Point 15**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 14**
W0.6	31:16	**Distortion Mesh Block 0 SU0 Search Point 13**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 12**
W0.5	31:16	**Distortion Mesh Block 0 SU0 Search Point 11**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 10**
W0.4	31:16	**Distortion Mesh Block 0 SU0 Search Point 9**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 8**
W0.3	31:16	**Distortion Mesh Block 0 SU0 Search Point 7**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 6**
W0.2	31:16	**Distortion Mesh Block 0 SU0 Search Point 5**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 4**
W0.1	31:16	**Distortion Mesh Block 0 SU0 Search Point 3**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 2**
W0.0	31:16	**Distortion Mesh Block 0 SU0 Search Point 1**
	15:0	**Distortion Mesh Block 0 SU0 Search Point 0** This is the distortion value at the location (0,0) within the search window. Each 256b (8 DW) message phase contains 16 distortion values for a given 4x4 search unit of the search window. Each word of a message phase contains a single 8x8 distortion at a given search point. The individual distortion values are arranged in raster-scan (row-major order) within the 4x4. For ShapeMode == 8, the first 4 message phases comprise the 1st SU of the 24x24 search window for all 4 block8x8_0-block8x8_1 distortions and are returned in raster-scan (row-major order). The next 4 message phases contain the 2nd SU and so on. Format = U16

Sample_8x8 State

This section contains different state definitions.

This state definition is used only by the *deinterlace* message. This state is stored as an array of up to 8 elements, each of which contains the dwords described here. The start of each element is spaced 8 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-7 that selects which element is being used is multiplied by 2 to determine the **Sampler Index** in the message descriptor.

SURFACE_STATE for Deinterlace, sample_8x8, and VME

This section contains media surface state definitions.

MEDIA_SURFACE_STATE

Restrictions: 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 for Sample_8x8 Message

SAMPLER_STATE has different formats, depending on the message type used. For SNB+, the sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

Sampler State
SAMPLER_STATE

- IEF Filter Type was dropped and is assumed to be Detailed filter
- IEF Filter Size was dropped and assumed to be 5x5.
- IEF Bypass – If we have Y/G-channel masked then the IEF bypass should always be forced to 1.
- Dword 0-15 is valid only when the function is “AVS Scaling”
- When IEF is Bypass, IEF ignores this state and need not store the state. AVS scaling is still on and uses this state. This would need to be indicated by the tag (maybe) returned so that AVS does not send the writes to IEF.
- Function Convolve : State is 144DWs
- Function MinMaxfilter/Erode/Dilate : State is 8DWs
- Function MinMax/BoolCentroid/Centroid : State is 0DWs. The sampler state is not required.
-

  This state definition is used only by the *sample_8x8* message only for specific function and the length of the state varies according to the function programmed in the message header.

For AVS and Convolve the state is stored as an array of up to 48 elements (**192** DWs), (upsize to 64 elements, 256DWs), each of which contains the dwords described here. The start of each of this state is spaced **256** dwords apart. The first element of the array is aligned to a 64-byte boundary. The **sampler index** in the message descriptor is multiplied by **32** to determine the offset from the base where the sampler state is to be read from. Sampler states with lower foot print than 32 elements should be packed at lower offsets and this sampler state for sample_8x8 message should be kept at the end. We can reuse existing sampler_index if the result of the multiplication of 32 is not overlapping with the existing states already programmed at the lower offsets. Two adjacent state of this type should have a space of 2 sample index.

For MinMaxFilter, Erode and Dilate the state is stored as an array of up to 2 elements (8 DWs), each of which contains the dwords as described in following section. The start of each of this state is spaced **8** dwords apart. The first element of the array is aligned to a 32-byte boundary. The **sampler index** in the message descriptor is multiplied by **2** to determine the offset from the base where the sampler state is to be read from. Sampler states with lower foot print than 2 elements should be packed at lower offsets and this sampler state for sample_8x8 message should be kept after it. Sampler states with larger footprint (192DWs) as described earlier should be packed after this. We can reuse existing sampler_index if the result of the multiplication of 2 is not overlapping with the existing states already programmed at the lower offsets. Two adjacent state of this type should have a space of 1 sample index.

Sampler State
SAMPLER_STATE_8x8_AVS
SAMPLER_STATE_8x8_AVS_COEFFICIENTS

Most of the existing functions are carried over and the states are programmed in exactly the same way. The following functions are added in SKL and use sampler state:

- Convolve (00000) & SKL_mode: 512 DWords or 128 Elements.

  Note: Convolve & BDW_mode is the same as for BDW with no change.
- 1D Vertical / Horizontal Convolution (function 01000 and 01001): 32 DWords or 8 Elements.
- 1 Pixel Convolution (Same as Convolve & BDW_mode): 144 DWords.
- AVS Scaling: 280 DWords or 70 elements. Occupies 128 Elements for each state.

From the above list we can see that we are required to support 128 Elements and 8 Elements which is new in SKL and the rest of element size are already supported in BDW. For this case the sampler_index is multiplied by 128 (for "2D Convolution and SKL_Mode" and "AVS Scaling") and 8 (for 1D convolution) respectively. Lower state sizes should use lower index and higher states sizes should use higher index and should be aligned to the sampler state size as in BDW.

Description
Function FloodFill/LBPCreation/CorrelationSearch: State is 0 DWords.
Function 1PixelConvolution: State is 192 DWords.
Function 1D Vertical/1D Horizontal Convolution: State is 32 DWords.
Function Convolution & SKL_mode: State is 512 DWords.
Function AVS Scaling: State is 280 DWords.

SIMD32/64 Messages

Initiating Message

SIMD32/64 Payload

SIMD32 Payload

DWord	Bits	Description
M1.7	31:0	**Reserved**
M1.6	31:0	**U 2nd Derivative** Defines the change in the delta U for adjacent pixels in the X direction. Format = IEEE_Float in normalized space.
M1.5	31:0	**Delta V**: defines the difference in V for adjacent pixels in the Y direction. **Programming Notes:** • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • This field is ignored for the deinterlace message type. • Negative **Delta V** are not supported and should be clamped to 0. Format = IEEE_Float in normalized space.
M1.4	31:0	**Delta U:** defines the difference in U for adjacent pixels in the X direction. **Programming Notes:** • **Delta U** multiplied by **Width** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • This field is ignored for the deinterlace message type. • Negative **Delta U** are not supported and should be clamped to 0. Format = IEEE_Float in normalized space.

DWord	Bits	Description
M1.3	31:0	**Pixel 0 V Address** Format: sample_unorm* : IEEE_Float in normalized space. Deinterlace: U32 (Range: [0,2046]) **:** Specifies the address for the pixel at the top left of the group and not the top of the message block sent in.
M1.2	31:0	**Pixel 0 U Address** Format: sample_unorm* : IEEE_Float in normalized space. Deinterlace: U32 (Range: [0,4095]) Specifies the address for the pixel at the top left of the group and not the top of the message block sent in.
M1.1	31:0	**Reserved**
M1.0	31:0	Ignored

SIMD64 Payload

IEF bypass has been moved from state to the payload. This needs to be forced to zero for all other functionality other than AVS. IEF bypass would not be controlled through inline parameter to the kernel for AVS with IEF off case. For Planar surfaces, the IEF bypass needs to be forced to zero for U,V channels by the kernel.

DWord	Bits	Description
M1.7	31:28	**Functionality** Bit 22 FunctionMode = 0:

Value	Function
0000b	AVS scaling
0001b	Convolve
0010b	MINMAX
0011b	MINMAXFILTER
0100b	ERODE
0101b	Dilate
0110b	BoolCentroid / BoolSum
0111b	Centroid
1000b	1D Vertical Convolution

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><td>1001b</td><td>1D Horizontal Convolution</td></tr>
<tr><td>1010b</td><td>1 Pixel Convolution</td></tr>
<tr><td>1011b</td><td>Flood Fill</td></tr>
<tr><td>1100b</td><td>LBP Creation</td></tr>
<tr><td>1101b</td><td>LBP Correlation</td></tr>
<tr><td>1110b</td><td>Reserved</td></tr>
<tr><td>1111b</td><td>Correlation Search</td></tr>
</table>
</td></tr>
<tr><td>M1.7</td><td>27</td><td>IEF Bypass
Ignored for all functions as it will always be IEF bypass except for AVS scaling operation.</td></tr>
<tr><td>M1.7</td><td>26:23</td><td>Control
[26:25] Message_Seq:

When Functionality is AVS:
11: 4x4 (only supported with Shuffle_OutputWriteback=1)
10: 16x8
01: 8x4 (only supported with Shuffle_OutputWriteback=1)
00: 16x4

When Functionality is Convolve/MinMaxFilter/LBP Correlation/LBP Creation:
11: 1x1 – This would be used in MinMaxFilter case & 1pixel convolve only for operation in scattered regions and don’t want to waste the power. This mode is not used and is reserved for 2D/1D Convolve, LBP Creation, and LBP Correlation.
10: 16x1 – This would be used in case we require the operation in scattered regions and don’t want to waste the throughput. 1-D Horizontal this will be 1x16. Not supported in LBP creation and LBP correlation.
0x: 16x4 – except for 1-D Horizontal this will be 4x16; illegal for 1 pixel convolve.

When Functionality is Erode/Dilate:
11: 32x1 – This would be used in case we require the operation in scattered regions and don’t want to waste the throughput.
10: 64x1 – This would be used in case we require the operation in scattered regions and don’t want to waste the throughput.</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>01: 32x4
00: 64x4
When Functionality is MINMAX/BoolCentroid/Centroid
1x: These modes should always force this to 1 since we don’t have SI sequencing.
0x: Unexpected behavior

When Functionality is FloodFill:
[26]: Reserved
[25]:
1 - 8-connected is used for floodfill.
0 - 4-connected is used for floodfill.

[24:23]:
When Functionality is MinMaxFilter/MinMax/LBP Creation Only:
11: Reserved
10: Disable Max Filter. (Min Filter only)/5x5 LBP
01: Disable Min Filter. (Max Filter only)/3x3 LBP
00: Both Min and Max filter/Both 3x3 and 5x5 LBP

[24:23]:
When Functionality is MinMaxFilter/LBP Creation Only:
11: Reserved
10: Disable Max Filter. (Min Filter only)/5x5 LBP
01: Disable Min Filter. (Max Filter only)/3x3 LBP
00: Both Min and Max filter/Both 3x3 and 5x5 LBP

When Functionality is Convolve:
<table><tr><th>Description</th></tr><tr><td>1x: Reserved</td></tr></table>Reserved for all other functionality.</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>M1.7</td><td>22</td><td>Reserved: MBZ</td></tr>
<tr><td>M1.7</td><td>21:0</td><td>Group ID Number

Used to group messages for reorder for sample_8x8 messages. All messages with the same Group ID must have the following in common: SURFACE_STATE, SAMPLER_STATE, Destination register on send instruction, M0, and M1 except for Horizontal and Vertical Block Number.</td></tr>
<tr><td>M1.6</td><td>31:0</td><td>Function = AVS:

U 2nd Derivative: Defines the change in the delta U for adjacent pixels in the X direction.

<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>SIMD64 Payload</td></tr><tr><td colspan="2">This field is ignored for messages other than sample_8x8.
(64 – (2*du))/35 >= ddu >= -du/18
2*ddu <= du</td></tr></table>
Format = IEEE_Float in normalized space

Function = FloodFill:

[31:16] Pixel mask of the bottom adjacent pixels in horizontal direction (Row 8 of 16x8 pixel).

[15:0] Pixel mask of the top adjacent pixels in horizontal direction (Row -1 of 16x8 pixel).

If function is not AVS or Floodfill, this field is ignored and assumed to be 0 (Reserved: MBZ).</td></tr>
<tr><td>M1.5</td><td>31:0</td><td>Function = AVS:

Delta V: defines the difference in V for adjacent pixels in the Y direction.

<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>SIM64 O</td></tr><tr><td colspan="2">The normalized value should be less than 1.
Delta V multiplied by Height in SURFACE_STATE must be less than 16 for the sample_8x8 message type.
This field is ignored for the deinterlace message type and VA sample_8x8 messages.
Negative Delta V is not supported and should be clamped to 0.</td></tr></table></td></tr>
</table>

DWord	Bits	Description
		Function = Correlation Search: [31:0] Normalized vertical origin of the reference image.%2

Vertical Block Number Restrictions

	Function	Msg Seq	Vblk Multiplier	Restrictions / Comments
Dword	M1.7	M1.7		Vertical block (vblk) is used for vertical offset except in 1D horizontal convolution where it is used as horizontal offset.
Bits	[31:28]	[26:25]		
	AVS Scaling	16x4	4	With Rotation of 90 and 270 the vblk should not be used and should be zero always.
		16x8	8	
		8x4		
		4x4	4	Vblk is supported with Rotation.
	Convolve	16x4	4	
		16x1	1	
	MinMax		0	Vblk is not used and should be zero.
	MinMaxFilter	16x4	4	
		16x1 or 1x1	1	
	Erode or Dilate	32x1 or 64x1	1	
		32x4 or 64x4	4	
	BoolCentroid		0	Vblk is not used and should be zero.
	Centroid		0	Vblk is not used and should be zero.
	1D Vertical Convolution	16x4	4	
		16x1	1	
	1D Horizontal Convolution	4x16	4	Vblk is used for horizontal offset rather than vertical in this function.
		1x16	1	
	1 Pixel Convolution	16x1, 1x1	0	Vblk is not used and should be zero.
	FloodFill	-	0	Vblk is not used and should be zero.
	LBP Creation	16x4	4	
	LBP Correlation	16x4	0	
	Correlation Search	-	0	Vblk is not used and should be zero.

Payload Parameter Definition

The table below shows all of the message types supported by the sampling engine. The **Message Type** field in the message descriptor determines which message is being sent. The **SIMD Mode** field determines the number of instances (i.e. pixels) and the formatting of the initiating and writeback messages. 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. The **Message Length** field is used to vary the number of parameters sent with each message. Higher-numbered parameters are optional, and default to a value of 0 if not sent but needed for the surface being sampled.

The message lengths are computed as follows, where "N" is the number of parameters ("N" is rounded up to the next multiple of 4 for SIMD4x2), and "H" is 1 if the header is present, 0 otherwise. The maximum message length allowed to the sampler is 11.

SIMD Mode	Message Length
SIMD4x2	H + (N/4)
SIMD8 SIMD8D	H + N
SIMD16	H + (2*N)

The response lengths are computed as follows:

SIMD Mode		Response Length Return Format = 32-bit	Response Length Return Format = 16-bit ***
SIMD4x2		1	not allowed
SIMD8	sample+killpix	5	not allowed
	all other message types	4	2 **
SIMD16		8 *	4 *

* For SIMD16, phases in the response length are reduced by 2 for each channel that is masked.

** : For SIMD8*, phases in the response length are reduced by 1 for each channel that is masked.

*** only

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.

: 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	Response Length	Alpha Write Channel Mask	Blue Write Channel Mask	Green Write Channel Mask	Red Write Channel Mask

SIMD Mode	Response Length	Alpha Write Channel Mask	Blue Write Channel Mask	Green Write Channel Mask	Red Write Channel Mask
SIMD8*	1	1	1	1	0
	2	1	1	0	0
	3	1	0	0	0
	4	0	0	0	0
SIMD16*	2	1	1	1	0
	4	1	1	0	0
	6	1	0	0	0
	8	0	0	0	0

: If **Pixel Fault Mask Enable** is enabled, response length must be set to a value one larger than those indicated in the tables above.

: Response Length of zero is allowed on all SIMD8* and SIMD16* sampler messages except sample+killpix, resinfo, sampleinfo, LOD, and gather4*. Header Present must be enabled and Pixel Null Mask Enable must be disabled. The Write Channel Mask for all four channels in the header must be 0 (channel enabled). A shader containing one or more of these messages is not allowed to send any render target read or write messages to the data port. When response length is set to zero, the following behavior occurs:

- When the sampler completes processing of the message, the resulting channels are delivered as input to the Render Cache Data Port in the form of a Render Target Write message without a header. Refer to the Data Port section for more details on this message.
- The fields normally set in the message descriptor for Render Target Write are set as follows:
  - End of Thread (EOT) comes from the EOT on the sampler message. These response length zero sampler messages are the only sampler messages allowed to have EOT enabled.
  - Last Render Target Select is set to true.
  - Slot Group Select is derived from EOT: If EOT is enabled, set to SLOTGRP_LO, otherwise SLOTGRP_HI.
  - RT Write Message Type is derived from the sampler's SIMD Mode field as follows: SIMD8 => SIMD8_LO, SIMD16 => SIMD16
  - Binding Table Index is set to the value set in the **Render Target Binding Table Index** field of the message header.

SIMD8D Messages (only):

Message Type	Mnemonic	Parameters										
		0	1	2	3	4	5	6	7	8	9	10
00000	sample	u0	u1	v0	v1	r	mlod					
00001	sample_b	bias	u0	u1	v0	v1	r	mlod				
00010	sample_l	lod	u0	u1	v0	v1	r					

Message Type	Mnemonic	Parameters										
		0	1	2	3	4	5	6	7	8	9	10
00011	sample_c	ref	u0	u1	v0	v1	r	mlod				
00100	sample_d	u0	u1	dudx	dudy	v0	v1	dvdx	dvdy	r	mlod	
00101	sample_b_c	ref	bias	u0	u1	v0	v1	r				
00110	sample_l_c	ref	lod	u0	u1	v0	v1	r				
01000	gather4	u0	u1	v0	v1	r						
01001	LOD	u0	u1	v0	v1							
10000	gather4_c	ref	u0	u1	v0	v1	r					
10001	gather4_po	u0	u1	v0	v1	offu	offv	r				
10010	gather4_po_c	ref	u0	u1	v0	v1	offu	offv	r			
10100	sample_d_c	ref	u0	u1	dudx	dudy	v0	v1	dvdx	dvdy	r	
10110	sample_min	u0	u1	v0	v1							
10111	sample_max	u0	u1	v0	v1							
11000	sample_lz	u0	u1	v0	v1	r						
11001	sample_c_lz	ref	u0	u1	v0	v1	r					

SIMD4x2 Messages

Message Type	Mnemonic	Parameters										
		0	1	2	3	4	5	6	7	8	9	10
00010	sample_l	u	v	r	ai	lod						
00100	sample_d	u	v	r	ai	dudx	dudy	dvdx	dvdy	drdx	drdy	mlod *
00110	sample_l_c	u	v	r	ai	ref	lod					
00111	ld	u	v	r	lod							
01000	gather4	u	v	r	ai							
01010	resinfo	lod										
01011	sampleinfo											
10000	gather4_c	u	v	r	ai	ref						
10001	gather4_po	u	v	r	ai	offu	offv					
10010	gather4_po_c	u	v	r	ref	offu	offv					
10100	sample_d_c	u	v	r	ai	dudx	dudy	dvdx	dvdy	drdx	drdy	ref
11100	ld2dms_w	u	v	r	lod *	si	mcsl	mcsh				
11101	ld_mcs	u	v	r	lod *							
11110	ld2dms	u	v	r	lod *	si	mcs					

* These parameters are allowed only for .

SIMD32/SIMD64 Messages

Message Type	mnemonic	Payload Layout	Message Length	Response Length
00000	sample_unorm	Pixel Shader	H + 1	8 **
00010	sample_unorm+killpix	Pixel Shader	H + 1	9 **
01000	deinterlace	Pixel Shader	H + 1	†
01100	sample_unorm	Media	H + 1	8 **
01010	sample_unorm+killpix	Media	H + 1	9 **
01011	sample_8x8	Media	H + 1	16 *
11111	cache_flush	no payload	1	1

* These parameters are allowed only for .

SIMD32/SIMD64 Messages

Message Type	mnemonic	Payload Layout	Message Length	Response Length
00000	sample_unorm	Pixel Shader	H + 1	8 **
00010	sample_unorm+killpix	Pixel Shader	H + 1	9 **
01000	deinterlace	Pixel Shader	H + 1	†
01100	sample_unorm	Media	H + 1	8 **
01010	sample_unorm+killpix	Media	H + 1	9 **
01011	sample_8x8	Media	H + 1	16 *
11111	cache_flush	no payload	1	1

* For sample_8x8, phases in the response length are reduced by 4 for each channel that is masked.

** For sample_unorm, phases in the response length are reduced by 2 for each channel that is masked.

† For deinterlace, response length depending on certain state fields. Refer to writeback message definition for details.

SIMD32_64 Message Descriptor

Please refer to the 3D Sampler Message Descriptor definition at **Message Descriptor - Sampling Engine** .

SIMD32_64 Message Header

Please refer to the 3D Sampler Message Header definition at Message Header.

Message Header

The message header for the sampling engine is the same regardless of the message type. If the header is not present, the behavior is as if the message was sent with all fields in the header set to zero (write channel masks are all enabled and offsets are zero). When Response length is 0 for sample_8x8 message

then the data from sampler is directly written out to memory using media write message. Message header needs to be present if mid-thread pre-emption is required.

DWord	Bits	Description
M0.5	31:5	Ignored
M0.5	4:0	**Output Format** Only for Sample_8x8 message with direct HDC write. 0 YCRCB_NORMAL 1 YCRCB_SWAPUVY 2 YCRCB_SWAPUV 3 YCRCB_SWAPY 4 PLANAR_420_8 (NV12 only) 5 PLANAR_Y8_UNORM 6 PLANAR_Y16_SNORM 7 Reserved 8 Reserved 9 Reserved 10 Reserved 11 Reserved 12 Reserved 13 Reserved 14 Reserved 15 Reserved 16 Reserved 17 PLANAR_Y32_UNORM 18 Reserved
M0.4	31:16	Destination Y Address (u16)
M0.4	15:0	Destination X Address in bytes (u16) Note that X Address should always be DWord-aligned.
M0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for "ld" and "resinfo" message types. This pointer is relative to the **Dynamic State Base Address**. Format = DynamicStateOfffset[31:5]
M0.3	4:1	Ignored
M0.3	0	Ignored

DWord	Bits	Description
M0.2	31:24	**Render Target or Destination Binding Table Index:** Specifies the index into the binding table for the render target or HDC for messages with response length of zero (the binding table index for the sampler surface is in the message descriptor). Format = U8 Range = [0,255]
M0.2	23	**Pixel Null Mask Enable:** Specifies whether the writeback message includes an extra phase indicating the pixel null mask. Refer to the "Writeback Message" section for details on format. This field must be disabled for sample+killpix and for all SIMD32/64 messages. Format = Enable Ignored for Sample_8x8 message
M0.2	22	**SIMD Mode Extension:** If SIMD Mode in the message descriptor is set to SIMD8D / SIMD4x2, this field specifies which mode is used. For other SIMD Modes, this field is ignored. 0: SIMD8D 1: SIMD4x2
M0.2	21	Ignored
M0.2	20	Ignored
M0.2	19:18	**SIMD32/64 Output Format Control** Specifies the output format of SIMD32/64 messages (sample_unorm* and sample_8x8). Ignored for other message types. Refer to the writeback message formats for details on how this field affects returned data. 0: 16 bit Full 1: 16 bit Chrominance Downsampled 2: 8 bit Full 3: 8 bit Chrominance Downsampled **Programming Note** **Context:** Message Header This field is ignored for sample_8x8 messages if the Function is not AVS and MinMaxFilter. For MinMaxFilter only 16bit Full and 8bit Full modes are supported. **Programming Note**

DWord	Bits	Description
		Context: Message Header This field is not used for HDC write messages.
M0.2	17:16	**Gather4 Source Channel Select:** Selects the source channel to be sampled in the gather4* messages. Ignored for other message types. 0: Red channel 1: Green channel 2: Blue channel 3: Alpha channel Note that for gather4*_c messages, this field must be set to 0 (Red channel).
M0.2	15	**Alpha Write Channel Mask:** Enables the alpha channel to be written back to the originating thread. 0: Alpha channel is written back. 1: Alpha channel is not written back. **Programming Note** **Context:** Message Header • A message with all four channels masked is not allowed. • This field is ignored for the deinterlace message. • This field must be set to zero for sample_8x8 in VSA mode. • This field must be set to zero for all gather4* messages. **Programming Note** **Context:** Message Header For Sample_8x8 messages, Alpha/Blue/Red channels should be always masked (set to 1) and only Green channel is enabled (set to 0).
M0.2	14	**Blue Write Channel Mask:** See Alpha Write Channel Mask.
M0.2	13	**Green Write Channel Mask:** See Alpha Write Channel Mask.
M0.2	12	**Red Write Channel Mask:** See Alpha Write Channel Mask.
M0.2	11:8	**U Offset:** The u offset from the _aoffimmi modifier on the “sample” or “ld” instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement. **Programming Note**

DWord	Bits	Description
		Context: Message Header • This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages. • This field is ignored if the "offu" parameter is included in the gather4* messages.
M0.2	7:4	**V Offset:** The v offset from the _aoffimmi modifier on the "sample" or "ld" instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement. **Programming Note** **Context:** Message Header • This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages. • This field is ignored if the "offu" parameter is included in the gather4* messages.
M0.2	3:0	**R Offset:** The r offset from the _aoffimmi modifier on the "sample" or "ld" instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement. This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
M0.1	31:0	Ignored
M0.0	31:0	Ignored

SIMD32_64 Payload Parameter Definition

Please refer to the 3D Sampler Payload Parameter Definition at Payload Parameter Definition.

SIMD32_64 Message Types

Please refer to the 3D Sampler Message Types definition at Message Types.

Writeback Message

SIMD32

Sample_unorm*

Pixels are numbered as follows:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

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 (using 16 bit Full mode as an example).

"16 bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 11 & 10 Red**
W0.4		**Pixel 9 & 8 Red**
W0.3		**Pixel 7 & 6 Red**
W0.2		**Pixel 5 & 4 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1.7		**Pixel 31 & 30 Red**
W1.6		**Pixel 29 & 28 Red**

DWord	Bit	Description
W1.5		**Pixel 27 & 26 Red**
W1.4		**Pixel 25 & 24 Red**
W1.3		**Pixel 23 & 22 Red**
W1.2		**Pixel 21 & 20 Red**
W1.1		**Pixel 19 & 18 Red**
W1.0		**Pixel 17 & 16 Red**
W2		**Pixels 15:0 Green**
W3		**Pixels 31:16 Green**
W4		**Pixels 15:0 Blue**
W5		**Pixels 31:16 Blue**
W6		**Pixels 15:0 Alpha**
W7		**Pixels 31:16 Alpha**

"16 Bit Chrominance Downsampled" Output Format Control Mode

In this mode the odd pixel red & blue channels are not included.

DWord	Bit	Description
W0.7	31:16	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 28 Red**
W0.6		**Pixel 26 & 24 Red**
W0.5		**Pixel 22 & 20 Red**

DWord	Bit	Description
W0.4		**Pixel 18 & 16 Red**
W0.3		**Pixel 14 & 12 Red**
W0.2		**Pixel 10 & 8 Red**
W0.1		**Pixel 6 & 4 Red**
W0.0		**Pixel 2 & 0 Red**
W1.7	31:16	**Pixel 15 Green**
	15:0	**Pixel 14 Green**
W1.6		**Pixel 13 & 12 Green**
W1.5		**Pixel 11 & 10 Green**
W1.4		**Pixel 9 & 8 Green**
W1.3		**Pixel 7 & 6 Green**
W1.2		**Pixel 5 & 4 Green**
W1.1		**Pixel 3 & 2 Green**
W1.0		**Pixel 1 & 0 Green**
W2.7		**Pixel 31 & 30 Green**
W2.6		**Pixel 29 & 28 Green**
W2.5		**Pixel 27 & 26 Green**
W2.4		**Pixel 25 & 24 Green**
W2.3		**Pixel 23 & 22 Green**
W2.2		**Pixel 21 & 20 Green**

DWord	Bit	Description
W2.1		**Pixel 19 & 18 Green**
W2.0		**Pixel 17 & 16 Green**
W3.7	31:16	**Pixel 30 Blue**
	15:0	**Pixel 28 Blue**
W3.6		**Pixel 26 & 24 Blue**
W3.5		**Pixel 22 & 20 Blue**
W3.4		**Pixel 18 & 16 Blue**
W3.3		**Pixel 14 & 12 Blue**
W3.2		**Pixel 10 & 8 Blue**
W3.1		**Pixel 6 & 4 Blue**
W3.0		**Pixel 2 & 0 Blue**
W4.7	31:16	**Pixel 15 Alpha**
	15:0	**Pixel 14 Alpha**
W4.6		**Pixel 13 & 12 Alpha**
W4.5		**Pixel 11 & 10 Alpha**
W4.4		**Pixel 9 & 8 Alpha**
W4.3		**Pixel 7 & 6 Alpha**
W4.2		**Pixel 5 & 4 Alpha**
W4.1		**Pixel 3 & 2 Alpha**
W4.0		**Pixel 1 & 0 Alpha**

DWord	Bit	Description
W5.7		**Pixel 31 & 30 Alpha**
W5.6		**Pixel 29 & 28 Alpha**
W5.5		**Pixel 27 & 26 Alpha**
W5.4		**Pixel 25 & 24 Alpha**
W5.3		**Pixel 23 & 22 Alpha**
W5.2		**Pixel 21 & 20 Alpha**
W5.1		**Pixel 19 & 18 Alpha**
W5.0		**Pixel 17 & 16 Alpha**

"8 Bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	**Pixel 31 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 30 Red**
	15:8	**Pixel 29 Red**
	7:0	**Pixel 28 Red**
W0.6		**Pixel 27:24 Red**
W0.5		**Pixel 23:20 Red**
W0.4		**Pixel 19:16 Red**
W0.3		**Pixel 15:12 Red**
W0.2		**Pixel 11:8 Red**

DWord	Bit	Description
W0.1		**Pixel 7:4 Red**
W0.0		**Pixel 3:0 Red**
W1		**Pixels 31:0 Green**
W2		**Pixels 31:0 Blue**
W3		**Pixels 31:0 Alpha**

"8 Bit Chrominance Downsampled" Output Format Control Mode

If either red or blue channel (but not both) are masked, the W0 register is included in the payload but the masked channel is not written to the GRF. If both are masked, W0 is not included in the payload (reducing the response length by one).

DWord	Bit	Description
W0.7	31:24	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	23:16	**Pixel 28 Red**
	15:8	**Pixel 26 Red**
	7:0	**Pixel 24 Red**
W0.6		**Pixel 22, 20, 18, 16 Red**
W0.5		**Pixel 14, 12, 10, 8 Red**
W0.4		**Pixel 6, 4, 2, 0 Red**
W0.3		**Pixel 30, 28, 26, 24 Blue**
W0.2		**Pixel 22, 20, 18, 16 Blue**
W0.1		**Pixel 14, 12, 10, 8 Blue**

DWord	Bit	Description
W0.0		**Pixel 6, 4, 2, 0 Blue**
W1.7	31:24	**Pixel 31 Green**
	23:16	**Pixel 30 Green**
	15:8	**Pixel 29 Green**
	7:0	**Pixel 28 Green**
W1.6		**Pixel 27:24 Green**
W1.5		**Pixel 23:20 Green**
W1.4		**Pixel 19:16 Green**
W1.3		**Pixel 15:12 Green**
W1.2		**Pixel 11:8 Green**
W1.1		**Pixel 7:4 Green**
W1.0		**Pixel 3:0 Green**
W2.7		**Pixel 31:28 Alpha**
W2.6		**Pixel 27:24 Alpha**
W2.5		**Pixel 23:20 Alpha**
W2.4		**Pixel 19:16 Alpha**
W2.3		**Pixel 15:12 Alpha**
W2.2		**Pixel 11:8 Alpha**
W2.1		**Pixel 7:4 Alpha**
W2.0		**Pixel 3:0 Alpha**

Additional Writeback Phase for sample_unorm+killpix

For the sample_unorm+killpix messages, an additional writeback phase is returned. The value of "n" depends on which channels are enabled for return and the **Output Format Control Mode**, this register will immediately follow the first part of the writeback message.

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>Wn.7:1</td><td></td><td>Reserved (not written)</td></tr>
<tr><td>Wn.0</td><td>31:0</td><td>Active Pixel Mask: 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.

The bits in this mask correspond to the pixels as follows and they are listed from upper left (MSB) lower right LSB:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td></tr>
<tr><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
</table>

Cache_flush

The writeback message is for cache_flush indicates that the flush has been completed. The destination register is not modified.

DWord	Bit	Description
W0.7:0		Reserved

Sample_8x8 Writeback Messages

The writeback message for sample_8x8 consists of up to 16 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 all four 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.

Pixels are numbered as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

"16 bit Full" Output Format Control Mode

DWord	Bits	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 11 & 10 Red**
W0.4		**Pixel 9 & 8 Red**
W0.3		**Pixel 7 & 6 Red**
W0.2		**Pixel 5 & 4 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1		**Pixel 31:16 Red**
W2		**Pixels 47:32 Red**
W3		**Pixels 63:33 Red**
W4		**Pixels 15:0 Green**
W5		**Pixels 31:16 Green**
W6		**Pixels 47:32 Green**
W7		**Pixels 63:33 Green**
W8		**Pixels 15:0 Blue**
W9		**Pixels 31:16 Blue**
W10		**Pixels 47:32 Blue**

DWord	Bits	Description
W11		**Pixels 63:33 Blue**
W12		**Pixels 15:0 Alpha**
W13		**Pixels 31:16 Alpha**
W14		**Pixels 47:32 Alpha**
W15		**Pixels 63:33 Alpha**

"16 Bit Chrominance Downsampled" Output Format Control Mode

In this mode the odd pixel red & blue channels are not included.

DWord	Bits	Description
W0.7	31:16	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 28 Red**
W0.6		**Pixel 26 & 24 Red**
W0.5		**Pixel 22 & 20 Red**
W0.4		**Pixel 18 & 16 Red**
W0.3		**Pixel 14 & 12 Red**
W0.2		**Pixel 10 & 8 Red**
W0.1		**Pixel 6 & 4 Red**
W0.0		**Pixel 2 & 0 Red**
W1.7		**Pixel 62 & 60 Red**
W1.6		**Pixel 58 & 56 Red**
W1.5		**Pixel 54 & 52 Red**

DWord	Bits	Description
W1.4		**Pixel 50 & 48 Red**
W1.3		**Pixel 46 & 44 Red**
W1.2		**Pixel 42 & 40 Red**
W1.1		**Pixel 38 & 36 Red**
W1.0		**Pixel 34 & 32 Red**
W2.7	31:16	**Pixel 15 Green**
	15:0	**Pixel 14 Green**
W2.6		**Pixel 13 & 12 Green**
W2.5		**Pixel 11 & 10 Green**
W2.4		**Pixel 9 & 8 Green**
W2.3		**Pixel 7 & 6 Green**
W2.2		**Pixel 5 & 4 Green**
W2.1		**Pixel 3 & 2 Green**
W2.0		**Pixel 1 & 0 Green**
W3		**Pixel 31:16 Green**
W4		**Pixel 47:32 Green**
W5		**Pixel 63:48 Green**
W6.7	31:16	**Pixel 30 Blue**
	15:0	**Pixel 28 Blue**
W6.6		**Pixel 26 & 24 Blue**
W6.5		**Pixel 22 & 20 Blue**

DWord	Bits	Description
W6.4		**Pixel 18 & 16 Blue**
W6.3		**Pixel 14 & 12 Blue**
W6.2		**Pixel 10 & 8 Blue**
W6.1		**Pixel 6 & 4 Blue**
W6.0		**Pixel 2 & 0 Blue**
W7.7		**Pixel 62 & 60 Blue**
W7.6		**Pixel 58 & 56 Blue**
W7.5		**Pixel 54 & 52 Blue**
W7.4		**Pixel 50 & 48 Blue**
W7.3		**Pixel 46 & 44 Blue**
W7.2		**Pixel 42 & 40 Blue**
W7.1		**Pixel 38 & 36 Blue**
W7.0		**Pixel 34 & 32 Blue**
W8.7	31:16	**Pixel 15 Alpha**
	15:0	**Pixel 14 Alpha**
W8.6		**Pixel 13 & 12 Alpha**
W8.5		**Pixel 11 & 10 Alpha**
W8.4		**Pixel 9 & 8 Alpha**
W8.3		**Pixel 7 & 6 Alpha**
W8.2		**Pixel 5 & 4 Alpha**
W8.1		**Pixel 3 & 2 Alpha**

DWord	Bits	Description
W8.0		**Pixel 1 & 0 Alpha**
W9		**Pixel 31:16 Alpha**
W10		**Pixel 47:32 Alpha**
W11		**Pixel 63:48 Alpha**

"8 Bit Full" Output Format Control Mode

DWord	Bits	Description
W0.7	31:24	**Pixel 31 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 30 Red**
	15:8	**Pixel 29 Red**
	7:0	**Pixel 28 Red**
W0.6		**Pixel 27:24 Red**
W0.5		**Pixel 23:20 Red**
W0.4		**Pixel 19:16 Red**
W0.3		**Pixel 15:12 Red**
W0.2		**Pixel 11:8 Red**
W0.1		**Pixel 7:4 Red**
W0.0		**Pixel 3:0 Red**
W1.7		**Pixel 63:60 Red**
W1.6		**Pixel 59:56 Red**
W1.5		**Pixel 55:52 Red**

DWord	Bits	Description
W1.4		**Pixel 51:48 Red**
W1.3		**Pixel 47:44 Red**
W1.2		**Pixel 43:40 Red**
W1.1		**Pixel 39:36 Red**
W1.0		**Pixel 35:52 Red**
W2		**Pixels 31:0 Green**
W3		**Pixels 63:32 Green**
W4		**Pixels 31:0 Blue**
W5		**Pixels 63:32 Blue**
W6		**Pixels 31:0 Alpha**
W7		**Pixels 63:32 Alpha**

"8 Bit Chrominance Downsampled" Output Format Control Mode

DWord	Bits	Description
W0.7	31:24	**Pixel 62 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 60 Red**
	15:8	**Pixel 58 Red**
	7:0	**Pixel 56 Red**
W0.6		**Pixel 54, 52, 50, 48 Red**
W0.5		**Pixel 46, 44, 42, 40 Red**
W0.4		**Pixel 38, 36, 34, 32 Red**

DWord	Bits	Description
W0.3		**Pixel 30, 28, 26, 24 Red**
W0.2		**Pixel 22, 20, 18, 16 Red**
W0.1		**Pixel 14, 12, 10, 8 Red**
W0.0		**Pixel 6, 4, 2, 0 Red**
W1.7	31:24	**Pixel 31 Green**
	23:16	**Pixel 30 Green**
	15:8	**Pixel 29 Green**
	7:0	**Pixel 28 Green**
W1.6		**Pixel 27:24 Green**
W1.5		**Pixel 23:20 Green**
W1.4		**Pixel 19:16 Green**
W1.3		**Pixel 15:12 Green**
W1.2		**Pixel 11:8 Green**
W1.1		**Pixel 7:4 Green**
W1.0		**Pixel 3:0 Green**
W2		**Pixel 63:32 Green**
W3.7	31:24	**Pixel 62 Blue**
	23:16	**Pixel 60 Blue**
	15:8	**Pixel 58 Blue**
	7:0	**Pixel 56 Blue**
W3.6		**Pixel 54, 52, 50, 48 Blue**

DWord	Bits	Description
W3.5		**Pixel 46, 44, 42, 40 Blue**
W3.4		**Pixel 38, 36, 34, 32 Blue**
W3.3		**Pixel 30, 28, 26, 24 Blue**
W3.2		**Pixel 22, 20, 18, 16 Blue**
W3.1		**Pixel 14, 12, 10, 8 Blue**
W3.0		**Pixel 6, 4, 2, 0 Blue**
W4.7	31:24	**Pixel 31 Alpha**
	23:16	**Pixel 30 Alpha**
	15:8	**Pixel 29 Alpha**
	7:0	**Pixel 28 Alpha**
W4.6		**Pixel 27:24 Alpha**
W4.5		**Pixel 23:20 Alpha**
W4.4		**Pixel 19:16 Alpha**
W4.3		**Pixel 15:12 Alpha**
W4.2		**Pixel 11:8 Alpha**
W4.1		**Pixel 7:4 Alpha**
W4.0		**Pixel 3:0 Alpha**
W5		**Pixel 63:32 Alpha**

Sampler_8x8 – Writeback Message for AVS

Output format when "Shuffle_OutputWriteback=1" for 8x4 and 16x4 messages

For 16x4 message :

MSB															LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48

For 8x4 message (though the message is only 32pixel, the below numbers are used to map to the same writeback format):

MSB							LSB
7	6	5	4	3	2	1	0
23	22	21	20	19	18	17	16
39	38	37	36	35	34	33	32
55	54	53	52	51	50	49	48

"16 bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:16	**Pixel 23 Red** **Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0)** **Range = [0000h:FF00h]**
	15:0	**Pixel 22 Red**
W0.6		**Pixel 21 & 20 Red**
W0.5		**Pixel 19 & 18 Red**
W0.4		**Pixel 17 & 16 Red**
W0.3		**Pixel 7 & 6 Red**
W0.2		**Pixel 5 & 4 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1.7		**Pixel 55 & 54 Red**
W1.6		**Pixel 53 & 52 Red**
W1.5		**Pixel 51 & 50 Red**
W1.4		**Pixel 49 & 48 Red**
W1.3		**Pixel 39 & 38 Red**
W1.2		**Pixel 37 & 36 Red**
W1.1		**Pixel 35 & 34 Red**
W1.0		**Pixel 33 & 32 Red**
W2		**Pixels 23:16, 7:0 Green**

DWord	Bit	Description
W3		**Pixels 55:48, 39:32 Green**
W4		**Pixels 23:16, 7:0 Blue**
W5		**Pixels 55:48, 39:32 Blue**
W6		**Pixels 23:16, 7:0 Alpha**
W7		**Pixels 55:48, 39:32 Alpha**
Below are valid only for 16x4 messages.		
W8		**Pixels 31:24, 15:8 Red**
W9		**Pixels 63:56, 47:40 Red**
W10		**Pixels 31:24, 15:8 Green**
W11		**Pixels 63:56, 47:40 Green**
W12		**Pixels 31:24, 15:8 Blue**
W13		**Pixels 63:56, 47:40 Blue**
W14		**Pixels 31:24, 15:8 Alpha**
W15		**Pixels 63:56, 47:40 Alpha**

"16 Bit Chrominance Downsampled" Output Format Control Mode

In this mode the odd pixel red & blue channels are not included.

DWord	Bit	Description
W0.7	31:16	**Pixel 54 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 52 Red**
W0.6		**Pixel 50 & 48 Red**
W0.5		**Pixel 38 & 36 Red**
W0.4		**Pixel 34 & 32 Red**
W0.3		**Pixel 22 & 20 Red**
W0.2		**Pixel 18 & 16 Red**
W0.1		**Pixel 6 & 4 Red**

DWord	Bit	Description
W0.0		**Pixel 2 & 0 Red**
W1.7	31:16	**Pixel 23 Green**
	15:0	**Pixel 22 Green**
W1.6		**Pixel 21 & 20 Green**
W1.5		**Pixel 19 & 18 Green**
W1.4		**Pixel 17 & 16 Green**
W1.3		**Pixel 7 & 6 Green**
W1.2		**Pixel 5 & 4 Green**
W1.1		**Pixel 3 & 2 Green**
W1.0		**Pixel 1 & 0 Green**
W2.7		**Pixel 55 & 54 Green**
W2.6		**Pixel 53 & 52 Green**
W2.5		**Pixel 51 & 50 Green**
W2.4		**Pixel 49 & 48 Green**
W2.3		**Pixel 39 & 38 Green**
W2.2		**Pixel 37 & 36 Green**
W2.1		**Pixel 35 & 34 Green**
W2.0		**Pixel 33 & 32 Green**
W3.7	31:16	**Pixel 54 Blue**
	15:0	**Pixel 52 Blue**
W3.6		**Pixel 50 & 48 Blue**

DWord	Bit	Description
W3.5		**Pixel 38 & 36 Blue**
W3.4		**Pixel 34 & 32 Blue**
W3.3		**Pixel 22 & 20 Blue**
W3.2		**Pixel 18 & 16 Blue**
W3.1		**Pixel 6 & 4 Blue**
W3.0		**Pixel 2 & 0 Blue**
W4.7	31:16	**Pixel 23 Alpha**
	15:0	**Pixel 22 Alpha**
W4.6		**Pixel 21 & 20 Alpha**
W4.5		**Pixel 19 & 18 Alpha**
W4.4		**Pixel 17 & 16 Alpha**
W4.3		**Pixel 7 & 6 Alpha**
W4.2		**Pixel 5 & 4 Alpha**
W4.1		**Pixel 3 & 2 Alpha**
W4.0		**Pixel 1 & 0 Alpha**
W5.7		**Pixel 55 & 54 Alpha**
W5.6		**Pixel 53 & 52 Alpha**
W5.5		**Pixel 51 & 50 Alpha**
W5.4		**Pixel 49 & 48 Alpha**
W5.3		**Pixel 39 & 38 Alpha**
W5.2		**Pixel 37 & 36 Alpha**

DWord	Bit	Description
W5.1		**Pixel 35 & 34 Alpha**
W5.0		**Pixel 33 & 32 Alpha**
Below DWs are only valid for 16x4 message		
W6.7	31:48	**Pixel 62 Red**
	15:0	**Pixel 60 Red**
W6.6		**Pixel 58 & 56 Red**
W6.5		**Pixel 46 & 44 Red**
W6.4		**Pixel 42 & 40 Red**
W6.3		**Pixel 30 & 28 Red**
W6.2		**Pixel 26 & 24 Red**
W6.1		**Pixel 14 & 12 Red**
W6.0		**Pixel 10 & 8 Red**
W7.7	31:48	**Pixel 63 Green**
	15:0	**Pixel 62 Green**
W7.6		**Pixel 61 & 60 Green**
W7.5		**Pixel 59 & 58 Green**
W7.4		**Pixel 57 & 56 Green**
W7.3		**Pixel 47 & 46 Green**
W7.2		**Pixel 45 & 44 Green**
W7.1		**Pixel 43 & 42 Green**
W7.0		**Pixel 41 & 40 Green**

DWord	Bit	Description
W8.7		**Pixel 31 & 30 Green**
W8.6		**Pixel 29 & 28 Green**
W8.5		**Pixel 27 & 26 Green**
W8.4		**Pixel 25 & 24 Green**
W8.3		**Pixel 15 & 14 Green**
W8.2		**Pixel 13 & 12 Green**
W8.1		**Pixel 11 & 10 Green**
W8.0		**Pixel 9 & 8 Green**
W9.7	31:48	**Pixel 62 Blue**
	15:0	**Pixel 60 Blue**
W9.6		**Pixel 58 & 56 Blue**
W9.5		**Pixel 46 & 44 Blue**
W9.4		**Pixel 42 & 40 Blue**
W9.3		**Pixel 30 & 28 Blue**
W9.2		**Pixel 26 & 24 Blue**
W9.1		**Pixel 14 & 12 Blue**
W9.0		**Pixel 10 & 8 Blue**
W10.7	31:48	**Pixel 63 Alpha**
	15:0	**Pixel 62 Alpha**
W10.6		**Pixel 61 & 60 Alpha**
W10.5		**Pixel 59 & 58 Alpha**

DWord	Bit	Description
W10.4		**Pixel 57 & 56 Alpha**
W10.3		**Pixel 47 & 46 Alpha**
W10.2		**Pixel 45 & 44 Alpha**
W10.1		**Pixel 43 & 42 Alpha**
W10.0		**Pixel 41 & 40 Alpha**
W11.7		**Pixel 31 & 30 Alpha**
W11.6		**Pixel 29 & 28 Alpha**
W11.5		**Pixel 27 & 26 Alpha**
W11.4		**Pixel 25 & 24 Alpha**
W11.3		**Pixel 15 & 14 Alpha**
W11.2		**Pixel 13 & 12 Alpha**
W11.1		**Pixel 11 & 10 Alpha**
W11.0		**Pixel 9 & 8 Alpha**

"8 Bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	**Pixel 55 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 54 Red**
	15:8	**Pixel 53 Red**
	7:0	**Pixel 52 Red**
W0.6		**Pixel 51:48 Red**

DWord	Bit	Description
W0.5		**Pixel 39:36 Red**
W0.4		**Pixel 35:32 Red**
W0.3		**Pixel 23:20 Red**
W0.2		**Pixel 19:16 Red**
W0.1		**Pixel 7:4 Red**
W0.0		**Pixel 3:0 Red**
W1		**Pixels [55:48],[39:32],[23:16],[7:0] Green**
W2		**Pixels [55:48],[39:32],[23:16],[7:0] Blue**
W3		**Pixels [55:48],[39:32],[23:16],[7:0] Alpha**
Below DWs are only valid for 16x4 message		
W4.7	31:24	**Pixel 63 Red**
	23:16	**Pixel 62 Red**
	15:8	**Pixel 61 Red**
	7:0	**Pixel 60 Red**
W4.6		**Pixel 59:56 Red**
W4.5		**Pixel 47:44 Red**
W4.4		**Pixel 43:40 Red**
W4.3		**Pixel 31:28 Red**
W4.2		**Pixel 27:24 Red**
W4.1		**Pixel 15:12 Red**
W4.0		**Pixel 11:8 Red**

DWord	Bit	Description
W5		**Pixels [63:56],[47:40],[31:24],[15:8] Green**
W6		**Pixels [63:56],[47:40],[31:24],[15:8] Blue**
W7		**Pixels [63:56],[47:40],[31:24],[15:8] Alpha**

"8 Bit Chrominance Downsampled" Output Format Control Mode

If either red or blue channel (but not both) are masked, the W0 and W3 registers are included in the payload but the masked channel is not written to the GRF. If both are masked, W0 and W3 are not included in the payload (reducing the response length by two).

DWord	Bit	Description
W0.7	31:24	**Pixel 54 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	23:16	**Pixel 52 Red**
	15:8	**Pixel 50 Red**
	7:0	**Pixel 48 Red**
W0.6		**Pixel 38, 36, 34, 32 Red**
W0.5		**Pixel 22, 20, 18, 16 Red**
W0.4		**Pixel 6, 4, 2, 0 Red**
W0.3		**Pixel 54, 52, 50, 48 Blue**
W0.2		**Pixel 38, 36, 34, 32 Blue**
W0.1		**Pixel 22, 20, 18, 16 Blue**
W0.0		**Pixel 6, 4, 2, 0 Blue**
W1.7	31:24	**Pixel 55 Green**
	23:16	**Pixel 54 Green**

DWord	Bit	Description
	15:8	**Pixel 53 Green**
	7:0	**Pixel 52 Green**
W1.6		**Pixel 51:48 Green**
W1.5		**Pixel 39:36 Green**
W1.4		**Pixel 35:32 Green**
W1.3		**Pixel 23:20 Green**
W1.2		**Pixel 19:16 Green**
W1.1		**Pixel 7:4 Green**
W1.0		**Pixel 3:0 Green**
W2.7		**Pixel 55:52 Alpha**
W2.6		**Pixel 51:48 Alpha**
W2.5		**Pixel 39:36 Alpha**
W2.4		**Pixel 35:32 Alpha**
W2.3		**Pixel 23:20 Alpha**
W2.2		**Pixel 19:16 Alpha**
W2.1		**Pixel 7:4 Alpha**
W2.0		**Pixel 3:0 Alpha**
Below DWs are only valid for 16x4 message		
W3.7	31:24	**Pixel 62 Red**
	23:16	**Pixel 60 Red**
	15:8	**Pixel 58 Red**

DWord	Bit	Description
	7:0	**Pixel 56 Red**
W3.6		**Pixel 46, 44, 42, 40 Red**
W3.5		**Pixel 30, 28, 26, 24 Red**
W3.4		**Pixel 14, 12, 10, 8 Red**
W3.3		**Pixel 62, 60, 58, 56 Blue**
W3.2		**Pixel 46, 44, 42, 40 Blue**
W3.1		**Pixel 30, 28, 26, 24 Blue**
W3.0		**Pixel 14, 12, 10, 8 Blue**
W4.7	31:24	**Pixel 63 Green**
	23:16	**Pixel 62 Green**
	15:8	**Pixel 61 Green**
	7:0	**Pixel 60 Green**
W4.6		**Pixel 59:56 Green**
W4.5		**Pixel 47:44 Green**
W4.4		**Pixel 43:40 Green**
W4.3		**Pixel 31:28 Green**
W4.2		**Pixel 27:24 Green**
W4.1		**Pixel 15:12 Green**
W4.0		**Pixel 11:8 Green**
W5.7		**Pixel 63:60 Alpha**
W5.6		**Pixel 59:56 Alpha**

DWord	Bit	Description
W5.5		**Pixel 47:44 Alpha**
W5.4		**Pixel 43:40 Alpha**
W5.3		**Pixel 31:28 Alpha**
W5.2		**Pixel 27:24 Alpha**
W5.1		**Pixel 15:12 Alpha**
W5.0		**Pixel 11:8 Alpha**

SIMD32 Surface State

Please refer to the 3D Surface State definition in the SURFACE_STATE topic.

SIMD32 Sampler State

Please refer to the 3D Sampler State definition at *Sampler State.*

3D Pipeline Stages

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

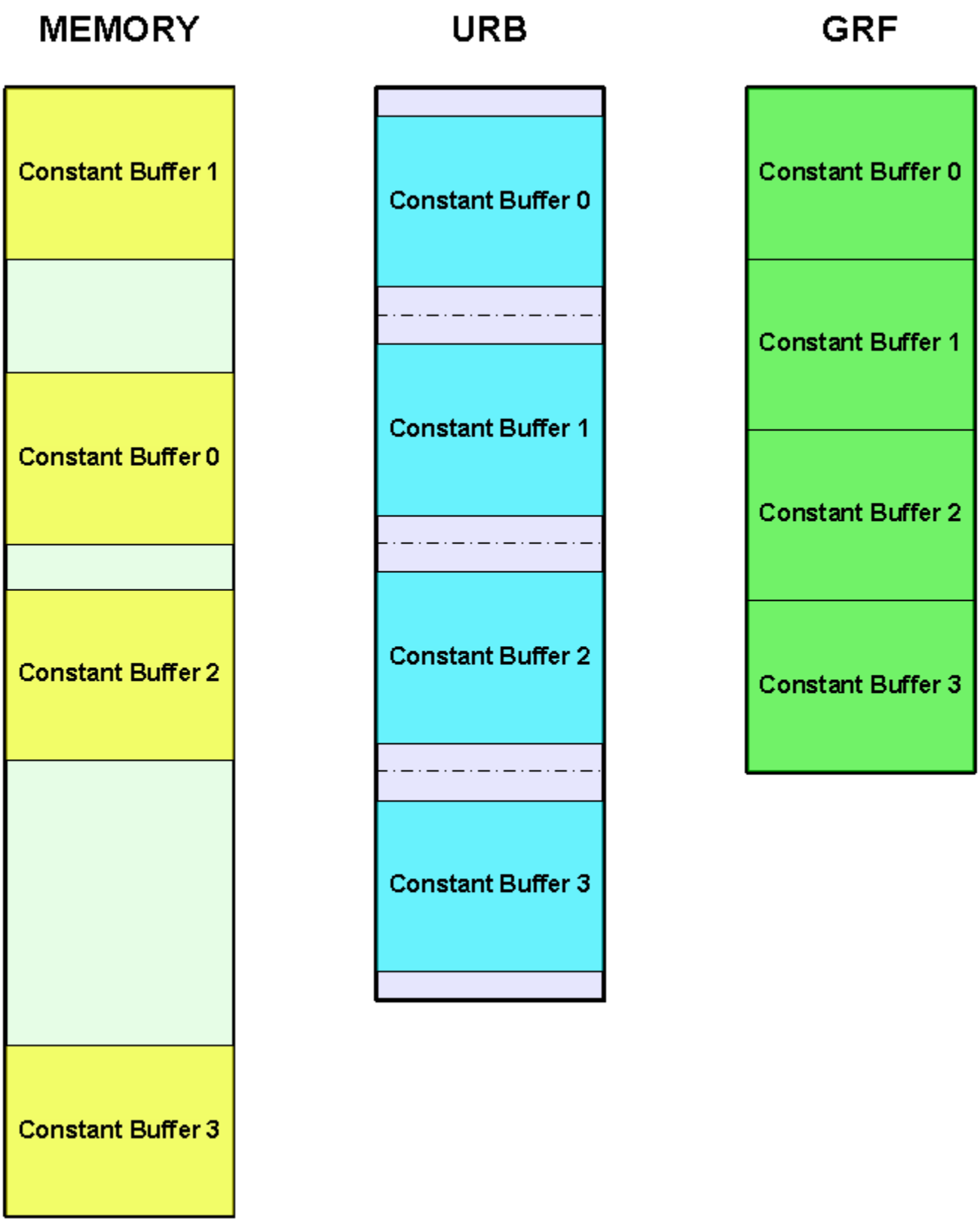
Pipeline Stage	Functions Performed
Command Stream (CS)	The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. Note that the CS stage is shared between the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

3D Pipeline-Level State

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

Programming Note	
Context:	3D Pipeline-Level State - Push Constant URB Allocation
The push constants are buffered in the Push Constant section of the URB which is part of the L3$. Software is required to program the hardware to allocate space in the URB for each shader push constant. The software is limited to the low addresses of the URB and must ensure that none of the shaders have overlapping handles. Software may use some if not all of the Push Constant region of the URB for pr-stage handle allocations as long as none of the push constants and handle allocations overlap. Refer to the various 3DSTATE_PUSH_CONSTANT_ALLOC_xx state commands for details regarding the maximum size of the Push Constant and other state programming information.	

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



The bubbles in the URB are caused by the constant buffer in memory starting on a half cacheline and

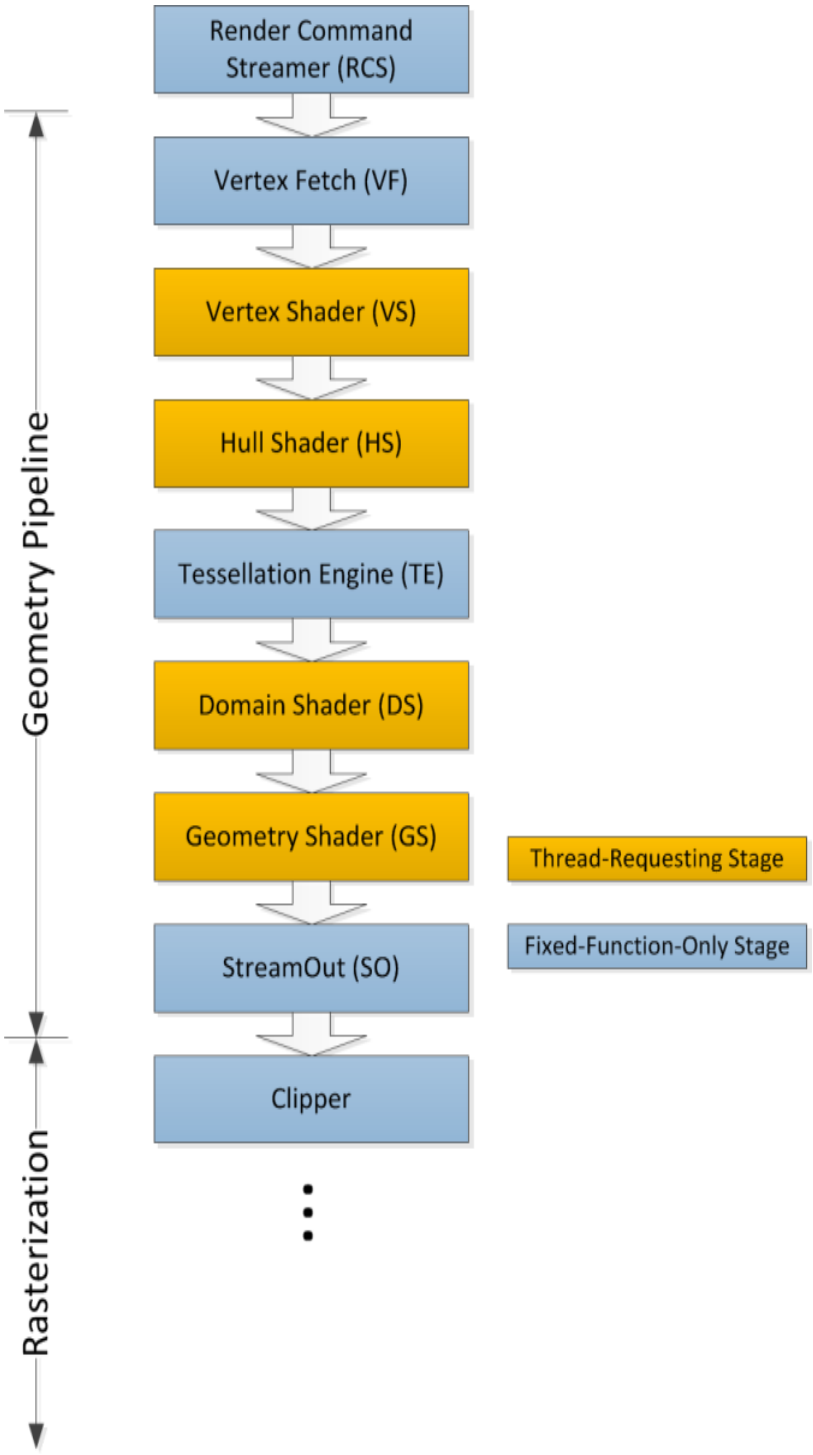
being an even number in length. If the constant buffer starts on an odd cacheline and has an odd number length then there will only be a bubble at the beginning of the buffer in the URB. If the constant buffer in memory starts on a cache line boundary and has an odd number length then the bubble will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

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

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.


Render Command Streamer (RCS)
Vertex Fetch (VF)
Vertex Shader (VS)
Hull Shader (HS)
Tessellation Engine (TE)
Domain Shader (DS)
Geometry Shader (GS)
StreamOut (SO)
Clipper
Geometry Pipeline
Rasterization
Thread-Requesting Stage
Fixed-Function-Only Stage

3D Primitives Overview

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

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

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

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

3D Primitive Topology Types

3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	• A list of independent line objects (2 vertices per line). • Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELIST_ADJ	• A list of independent line objects with adjacency information (4 vertices per line). • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
LINELOOP	• Similar to a 3DPRIM_LINESTRIP, though the last vertex is connected back to the

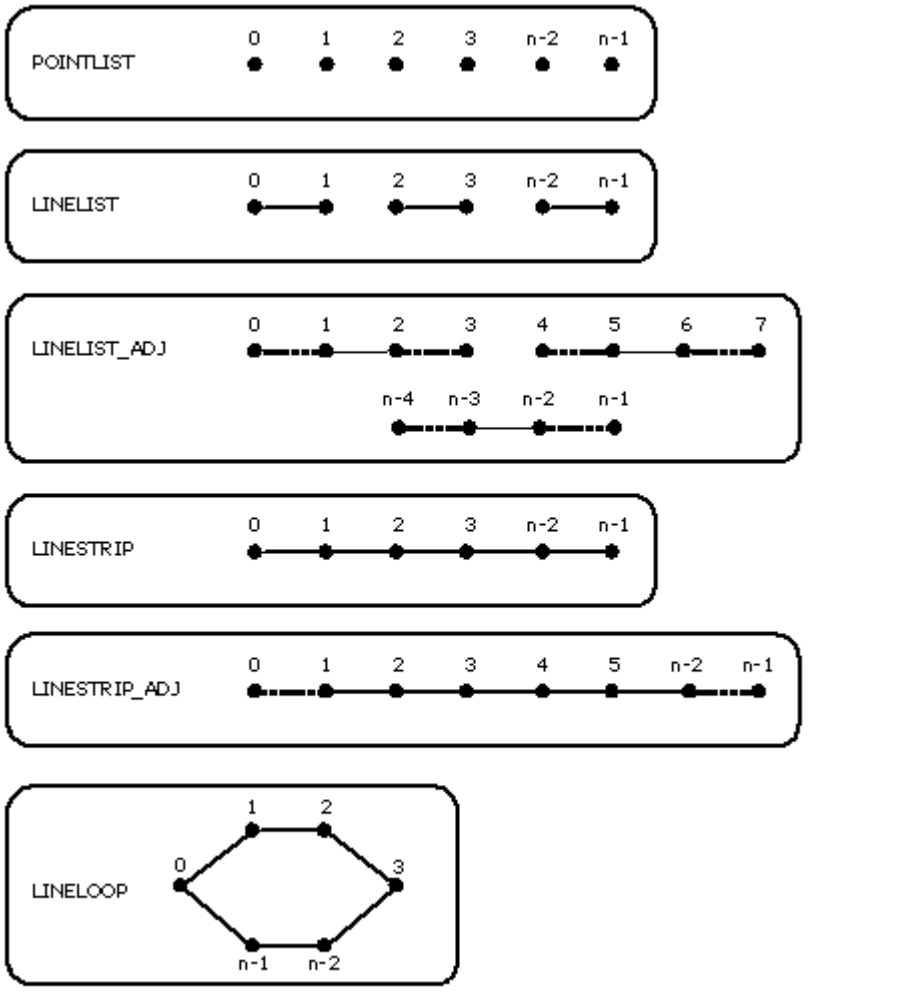
3D Primitive Topology Type (ordered alphabetically)	Description
	initial vertex via a line object. The LINELOOP topology is converted to LINESTRIP topology at the beginning of the 3D pipeline. • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored. (The 2-vertex case is required by OGL). • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
LINESTRIP	• A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define a connected line object. • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored.
LINESTRIP_ADJ	• A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define connected line object. The first and last segments are adjacent–only vertices. • Normal usage expects at least 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
LINESTRIP_BF	• Similar to LINESTRIP, except treated as “backfacing’ during rasterization (stencil test). • This can be used to support “line” polygon fill mode when two-sided stencil is enabled.
LINESTRIP_CONT	• Similar to LINESTRIP, except LineStipple (if enabled) is continued (vs. reset) at the start of the primitive topology. • This can be used to support line stipple when the API-provided primitive is split across multiple tolopologies.
LINESTRIP_CONT_BF	Combination of LINESTRIP_BF and LINESTRIP_CONT variations.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	• Similar to POINTLIST, except treated as “backfacing’ during rasterization (stencil test). • This can be used to support “point” polygon fill mode when two-sided stencil is enabled.
POLYGON	• Similar to TRIFAN, though the first vertex always provides the “flat-shaded” values (vs. this being programmable through state). • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.

3D Primitive Topology Type (ordered alphabetically)	Description
QUADLIST	• A list of independent quad objects (4 vertices per quad). • The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline. • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored.
QUADSTRIP	• A list of vertices connected such that, after the first two vertices, each additional pair of vertices are associated with the previous two vertices to define a connected quad object. • Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored.
RECTLIST	• A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle. V0=LowerRight, V1=LowerLeft, V2=UpperLeft. Implied V3 = V0-V1+V2. • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored. • The RECTLIST primitive is supported 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: • 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).
RECTLIST_SUBPIXEL	The subpixel precise, axis-aligned bounding box of the object's 3 vertices is rendered.
TRIFAN	• Triangle objects arranged in a fan (or polygon). The initial vertex is maintained as a common vertex. After the second vertex, each additional vertex is associated with the previous vertex and the common vertex to define a connected triangle object. • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRIFAN_NOSTIPPLE	• Similar to TRIFAN, but poylgon stipple is not applied (even if enabled). • This can be used to support "point" polygon fill mode, under the combination

3D Primitive Topology Type (ordered alphabetically)	Description
	of the following conditions: (a) when the frontfacing and backfacing polygon fill modes are different (so the final fill mode is not known to the driver), (b) one of the fill modes is “point” and the other is “solid”, (c) point mode is being emulated by converting the point into a trifan, (d) polygon stipple is enabled. In this case, polygon stipple should not be applied to the points-emulated-as-trifans.
TRILIST	• A list of independent triangle objects (3 vertices per triangle). • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.
TRILIST_ADJ	• A list of independent triangle objects with adjacency information (6 vertices per triangle). • Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
TRISTRIP	• A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRISTRIP_ADJ	• A list of vertices where the even-numbered (including 0th) vertices are connected such that, after the first two vertex pairs, each additional even-numbered vertex is associated with the last two even-numbered vertices to define a connected triangle object. The odd-numbered vertices are adjacent-only vertices. VFUNIT will complete a drawcall with the topology of tristrip_adj even if there is a preemption request in the middle of the draw call. • Normal usage expects at least 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
TRISTRIP_REVERSE	Similar to TRISTRIP, though the sense of orientation (winding order) is reversed – this allows SW to break long tristrips into smaller pieces and still maintain correct face orientations.

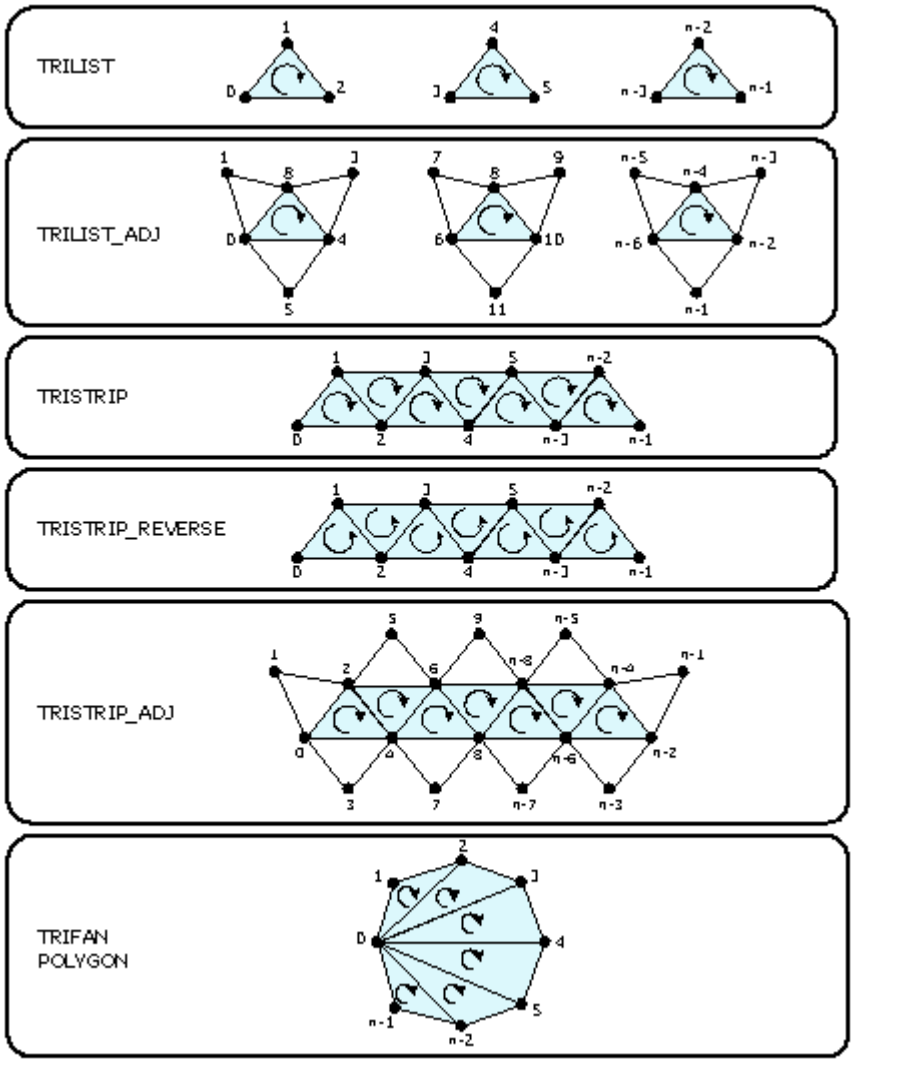
3D Primitive Topology Type (ordered alphabetically)	Description
: PATCHLIST_n	List of n-vertex “patch” objects. These topologies cannot be rendered directly – the tessellation units must be used to convert them into points, lines, or triangles to produce rasterization results. (VS, GS, and StreamOutput operations can also be performed.)

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

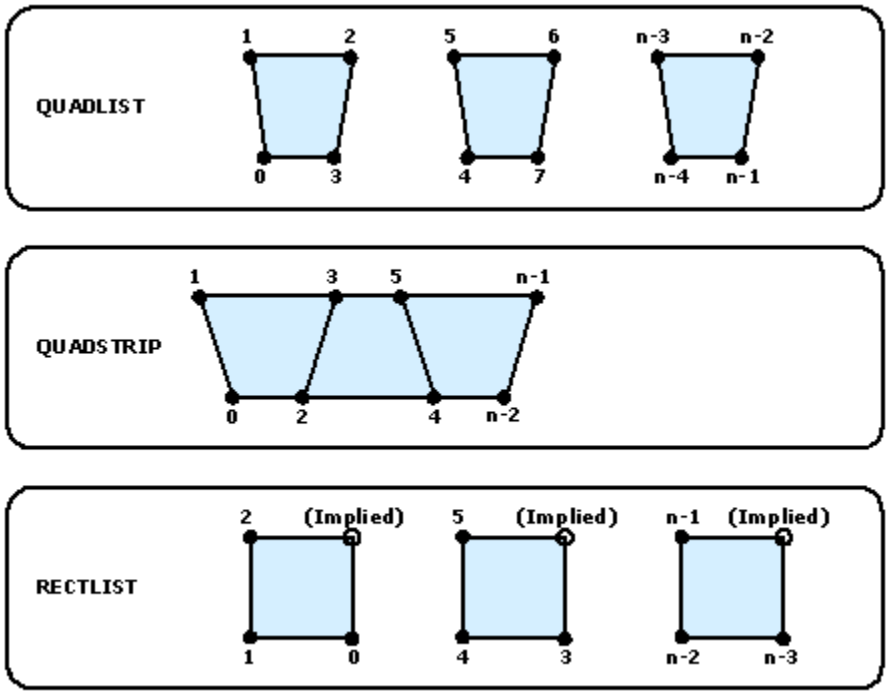


B6815-01

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


TRILIST
TRILIST_ADJ
TRISTRIP
TRISTRIP_REVERSE
TRISTRIP_ADJ
TRIFAN
POLYGON

B6816-01


QUADLIST
1
2
0
3
5
6
4
7
n-3
n-2
n-4
n-1
QUADSTRIP
1
3
5
n-1
0
2
4
n-2
RECTLIST
2
(Implied)
1
0
5
(Implied)
4
3
n-1
(Implied)
n-2
n-3

B6818-01

Thread Request Generation

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

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

Thread Control Information

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

State Variables Included in Thread Control Information

State Variable	Usage	FFs
Kernel Start Pointer	This field, together with the **General State Pointer**, specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Pointer**.	All FFs spawning threads
GRF Register Block Count	Specifies, in 16-register blocks, how many GRF registers are required to run the kernel. The Thread Dispatcher will only seek candidate EUs that have a sufficient number of GRF register blocks available. Upon selecting a target EU, the Thread DIspatcher will generate a logical-to-physical GRF mapping and provide this to the target EU.	All FFs spawning threads
Single Program Flow (SPF)	Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*.	All FFs spawning threads
Thread Dispatch Priority	The Thread Dispatcher will give priority to those thread requests with Thread Dispatch Priority of HIGH_PRIORITY over those marked as LOW_PRIORITY. Within these two classes of thread requests, the Thread Dispatcher applies a priority order (e.g., round-robin --- though this algorithm is considered a device implementation-dependent detail).	All FFs spawning threads
Floating	This determines the initial value of the **Floating Point Mode** bit of the EU's CR0	All FFs spawning

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

Thread Payload 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 GEN4 subsystem. The FF unit uses this field (when transmitted in a Message Header to the URB Function) to detect messages emanating from its spawned threads.	All FFs spawning threads
Snapshot Flag		All FFs spawning threads
Thread ID	This field uniquely identifies this thread within the FF unit over some period.	All FFs spawning threads
Scratch Space Pointer	This is the starting location of the thread's allocated scratch space, specified as an offset from the **General State Base Address**. Note that scratch space is allocated by the FF unit on a per-thread basis, based on the **Scratch Space Base Pointer** and **Per-Thread Scratch Space Size** state variables. FF units assign a thread an arbitrarily-positioned region within this space. The scratch space for multiple (API-visible) entities (vertices, pixels) is interleaved within the thread's scratch space.	All FFs spawning threads
Dispatch ID	This field identifies this thread within the outstanding threads spawned by the FF unit. This field does *not* uniquely identify the thread over any significant period. **Implementation Note:** This field is effectively an "active thread index". It is used on a thread's URB allocation request to identify which thread's handle pool is to source the allocation. It is used upon thread termination to free up the thread's scratch space allocation.	All FFs spawning threads
Binding Table Pointer	This field, together with the **Surface State Base Pointer**, specifies the starting location of the Binding Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **Surface State Base Pointer**. See *Shared Functions* for a description of a Binding Table.	All FFs spawning threads

Fixed Payload Header Field (non-FF-specific)	Description	FFs
Sampler State Pointer	This field, together with the **General State Base Pointer**, specifies the starting location of the Sampler State Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **General State Base Pointer**. See *Shared Functions* for a description of a Sampler State Table.	**All FFs spawning threads which sample (VS, GS, WM)**
Per Thread Scratch Space	This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.	All FFs spawning threads
Handle ID <*n*>	This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS, GS, CLIP, SF
URB Return Handle <*n*>	This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS, GS, CLIP, SF
Primitive Topology Type	As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (for example, for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread. GEN4 kernels written to process different types of objects can use this value to direct that processing. E.g., when a CLIP kernel is to provide clipping for all the various primitive types, the kernel would need to examine the Primitive Topology Type to distinguish between point, lines, and triangle clipping requests. **Note:** In general, this field is identical to the Primitive Topology Type associated with the primitive vertices as received by the FF unit. Refer to the individual FF unit chapters for cases where the FF unit modifies the value before passing it to the thread. (for example, certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).	GS, CLIP, SF, WM

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.

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 to the GEN4 subsystem. The subsystem will read the appropriate 256-bit locations of the URB, optionally perform swizzling (VS only), and write the results into sequential GRF registers (starting at **Dispatch GRF Start Register for URB Data**).

State Variables Controlling Payload URB Data

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

Programming Restrictions: (others may already been mentioned)

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

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, processed by CLIP threads, 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, and CLIP threads can compute, format, and write new VUEs as thread output.

There are only two 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.

- Just prior to the CLIP stage, all VUEs are read-back: Optional readback of ClipDistance values (up to 8 floats in an aligned 256-bit URB row).
- Just after the CLIP stage, on clip-generated VUEs are read-back: Readback of the Vertex Header (first 256 bits of the VUE).

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.
- (There is a pathological case where the CLIP state can be programmed to guarantee that all CLIP-incoming vertices are consumed – regardless of the data read back prior to the CLIP stage – and therefore only the CLIP thread-generated vertices would require Vertex Headers.)

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	**Render Target Array Index (RTAIndex).** This value is (eventually) used to index into a specific element of an "array" Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required, software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a 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. Software can force an RTAIndex of 0 to be used (effectively ignoring the setting of this DWord) by use of the **ForceZeroRTAIndex** bit (3DSTATE_CLIP). Otherwise the read-back value will be used to select an RTArray element, after being clamped to the RTArray surface's [**MinimumArrayElement, Depth**] range (SURFACE_STATE). Format: 0-based U32 index value
D2	31:0	**Viewport Index.** This value is used to select one of a possible 16 sets of viewport (VP) state parameters in the Clip unit's VertexClipTest function and in the SF unit's ViewportMapping and Scissor functions. The GS unit (even if disabled) will read back this value for all vertices exiting the GS stage and entering the Clip stage. When enabled, the GS unit will range-check the value against [0,**Maximum VPIndex**] (see GS_STATE, CLIP_STATE). After this range-check the values are sent down the pipeline and used in the Clip unit's VertexClipTest function. For vertices passing through the Clip stage, these values will also be sent to the SF unit for use in ViewportMapping and Scissor functions. The Clip unit (if enabled) will read back this value only for vertices generated by CLIP threads. The Clip unit will perform a range clamp similar to the GS unit. Software can force a value of 0 to be used by programming **Maximum VPIndex** to 0.

DWord	Bits	Description
		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 both the GS and Clip units. Format: FLOAT32
D4	31:0	**Vertex Position X Coordinate.** This field contains the X component of the vertex's 4D space position. Format: FLOAT32
D5	31:0	**Vertex Position Y Coordinate.** This field contains the Y component of the vertex's 4D space position. Format: FLOAT32
D6	31:0	**Vertex Position Z Coordinate.** This field contains the Z component of the vertex's NDC space position. Format: FLOAT32
D7	31:0	**Vertex Position W Coordinate.** This field contains the Z component of the vertex's 4D space position. Format: FLOAT32
D8	31:0	**ClipDistance 0 Value (optional).** If the **UserClipDistance Clip Test Enable Bitmask** bit (3DSTATE_CLIP) is set, this value will be read from the URB in the Clip stage. If the value is found to be less than 0 or a NaN, the vertex's UCF<0> bit will set in the Clip unit's VertexClipTest function. If the **UserClipDistance Clip Test Enable Bitmask** bit is clear, this value will not be read back, and the vertex's UCF<0> bit will be zero by definition. Format: FLOAT32 ClipDistance Values are enabled for clip/cull test in the Clip stage in one of two modes: Normally the corresponding Enable Bitmasks are obtained from the state programmed in the last "vertex-producing" stage (VS/DS/GS) that is enabled prior to the Clip stage. E.g., if VS and DS are enabled but GS is disabled, the masks are obtained from 3DSTATE_DS. Alternatively, the Enable Bitmasks can be obtained directly from corresponding masks programmed via 3DSTATE_CLIP, through use of 3DSTATE_CLIP's Force User Clip Distance [Cull/Clip] Test Enable Bitmask state bits (see description of 3DSTATE_CLIP).
D9	31:0	**ClipDistance 1 Value (optional).** See above.

DWord	Bits	Description
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	**(Remainder of Vertex Elements).** The absolute maximum size limit on this data is specified via a maximum limit on the amount of data that can be read from a VUE (including the Vertex Header) (**Vertex Entry URB Read Length** has a maximum value of 63 256-bit units). Therefore the Remainder of Vertex Elements has an absolute maximum size of 62 256-bit units. Of course the actual allocated size of the VUE can and will limit the amount of data in a VUE.

Vertex Positions

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

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

CLIP unit uses the **3DSTATE_CLIP.PerspectiveDivideDisable** bit to deteremine 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.

In , 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.

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 were 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_LINELIST_ADJ	Y
3DPRIM_LINESTRIP_ADJ	Y
3DPRIM_TRILIST_ADJ	Y
3DPRIM_TRISTRIP_ADJ	Y
3DPRIM_TRISTRIP_REVERSE	Y

Definition	Cut Index?
3DPRIM_RECTLIST	N
3DPRIM_POINTLIST_BF	Y
3DPRIM_LINESTRIP_CONT	Y
3DPRIM_LINESTRIP_BF	Y
3DPRIM_LINESTRIP_CONT_BF	Y
3DPRIM_TRIFAN_NOSTIPPLE	N

3DSTATE_INDEX_BUFFER

Vertex Buffers (VB) State

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

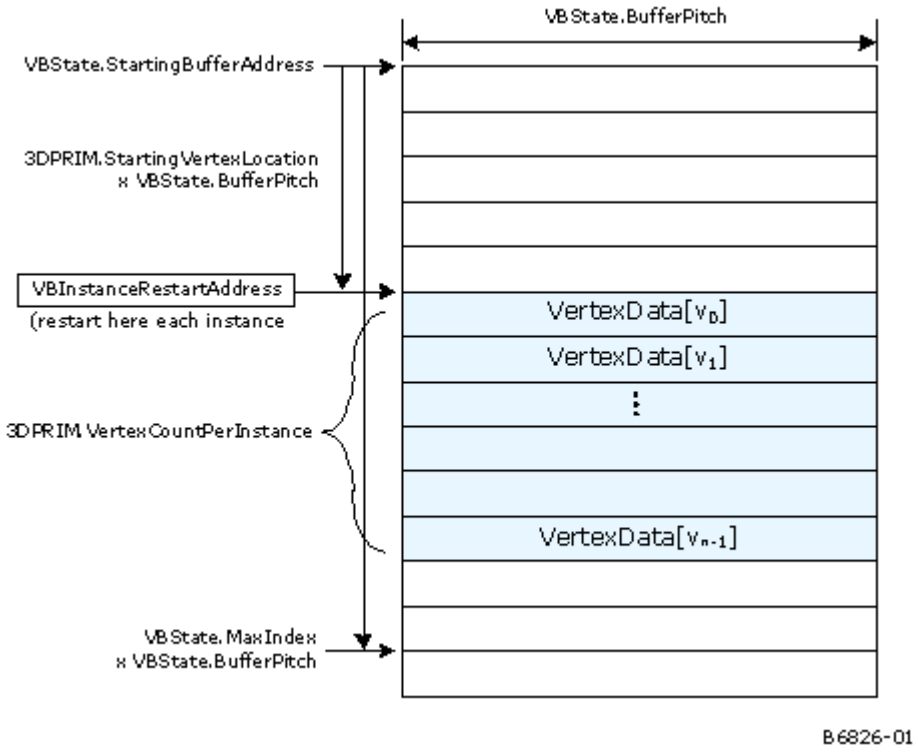
Most input vertex data is sourced from memory-resident VBs. A VB is a 1D array of structures, where the size of the structure as defined by the VB's **BufferPitch**. VBs are accessed either as *VERTEXDATA buffers* or *INSTANCEDATA buffers*, as defined by the **InstancingEnable** state in 3DSTATE_VF_INSTANCING.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

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

Register
3DSTATE_VERTEX_BUFFERS
VERTEX_BUFFER_STATE

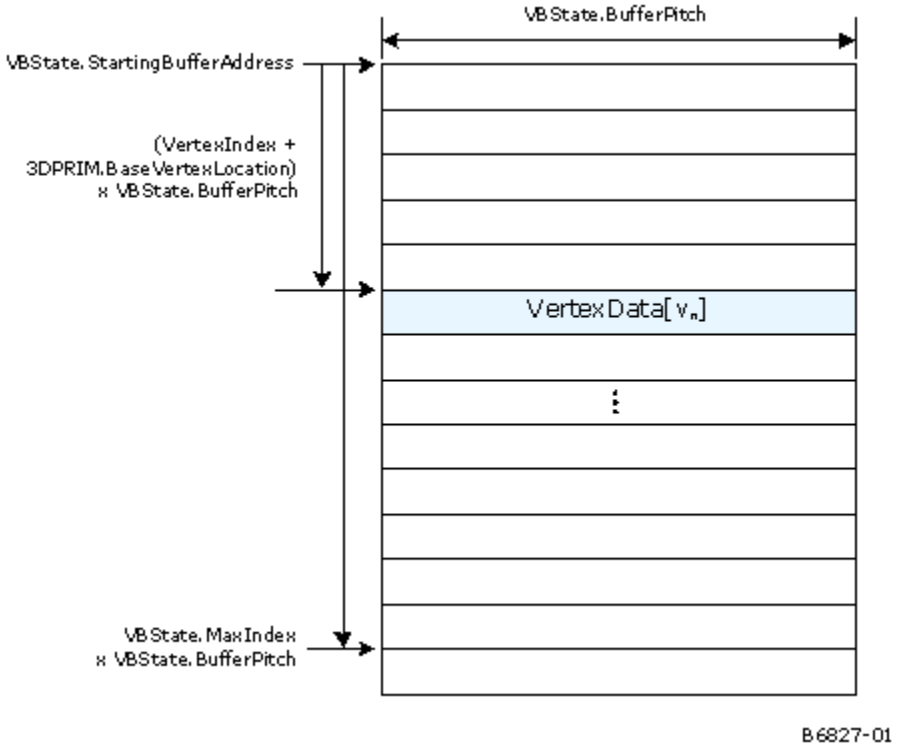
VERTEXDATA Buffers – SEQUENTIAL Access

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



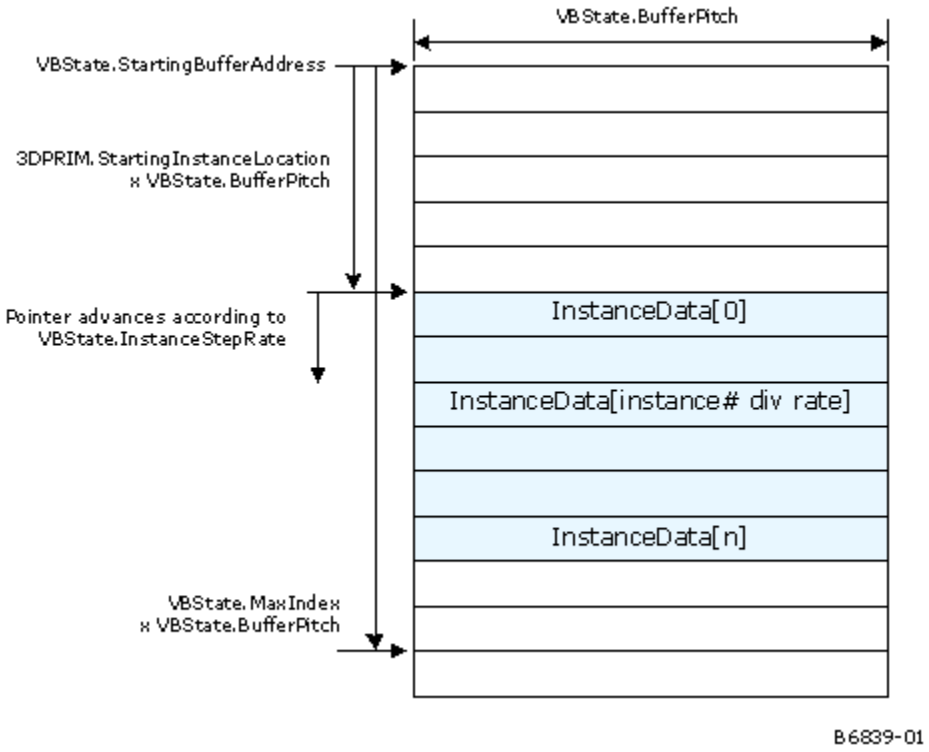
VERTEXDATA Buffers – RANDOM Access

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



INSTANCEDATA Buffers

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



Vertex Definition State

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

Input Vertex Definition

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

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

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

Register
VERTEX_ELEMENT_STATE
3DSTATE_VERTEX_ELEMENTS
3D_Vertex_Component_Control
3DSTATE_VF_INSTANCING
3DSTATE_VF_SGVS
3DSTATE_VF_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.)

Modern 3D APIs include the notion of native support for *instanced geometry*. The premise is that the application models a high-level object, and then proceeds to process (render) multiple instances of the object, with each instance including some modification to the data associated with the object. For example, a single chair object could be modeled and rendered multiple times, with each instance using a different position transformation (to place several chairs around a table). That transformation matrix would be considered *instance data*, in that it would be fixed for each instance of the object, changing only between instances. This is opposed to the *vertex data*, which could be unique for each vertex of the object. Note that the instance data is replicated at each vertex as part of the Input Assembly process, and so could also be considered "vertex data" in that respect. In fact, the notion of instance data is confined to the VF stage, and the remainder of the 3D pipeline is unaware of the distinction.

Although the ability to perform instanced geometry ("manually") was available in legacy APIs, modern APIs add support for a single Draw() call to process multiple instances. The application specifies the number of instances, and the number of vertices in the instanced object (instance) in addition to other

parameters discussed later. The Draw() support effectively becomes a nested loop, with the outer loop dealing with instance variables, and the inner loop sequencing through the object vertices.

Given the flexibility and programmability of the 3D pipeline, the division between vertex and instance data is arbitrary. The application simply marks input data buffers as either containing vertex or instance data. In the outer instance loop, the data specified as coming from *instance buffers* is (effectively) fetched/generated and then associated with each vertex of the instance in the inner loop and combined with the vertex data. New instance data is then gathered for the next instance (the "stepping" of instance data is specified on a per-buffer basis, and different portions of the instance data are allowed to step at different rates -- refer to details below).

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 GEN 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.

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

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

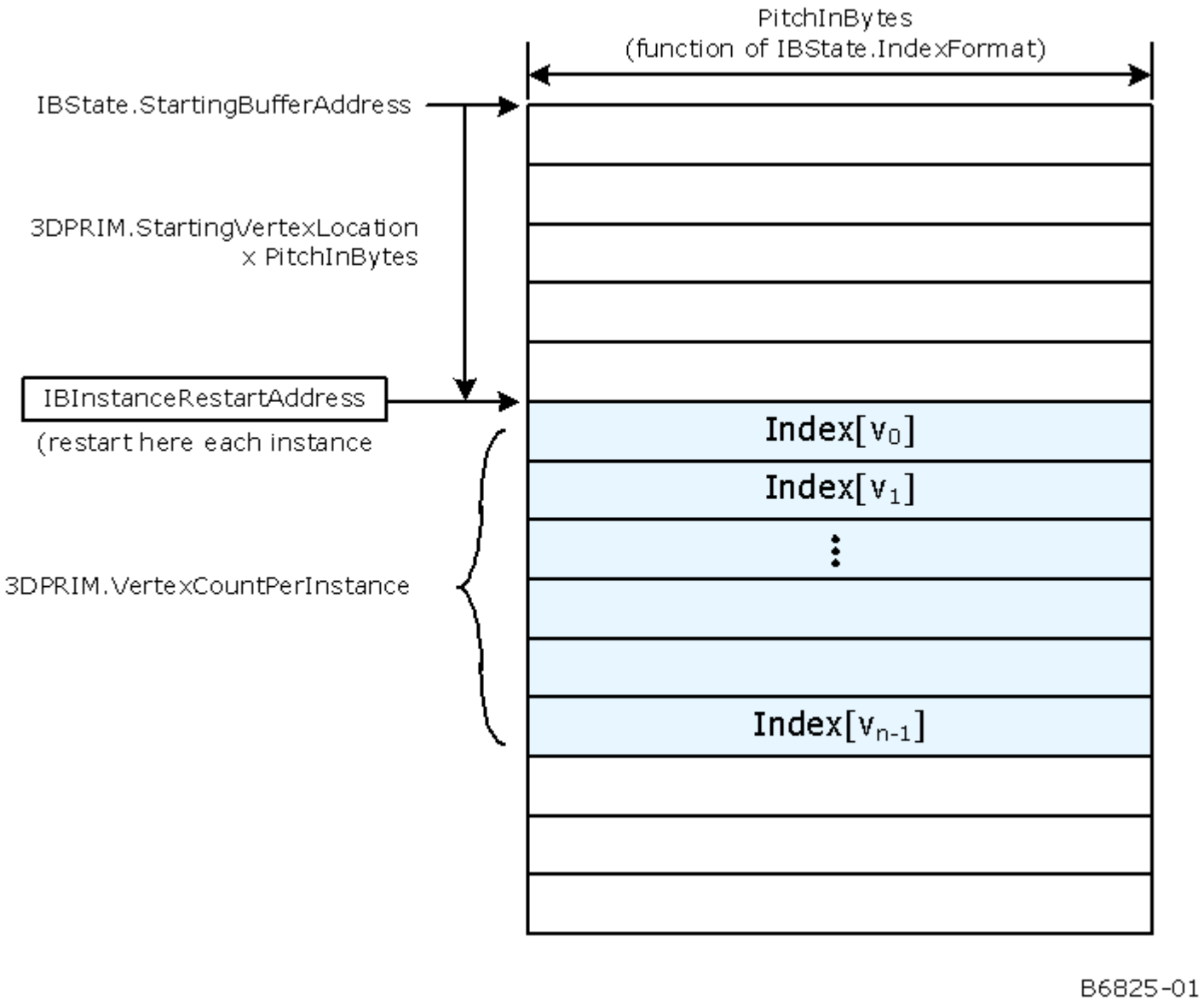
The optional insertion of VertexID and/or InstanceID into the input vertex data occurs as a separate step after the processing of VERTEX_ELEMENT structures, and is controlled via the 3DSTATE_VF_SGVS command.

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

3D Primitive Processing

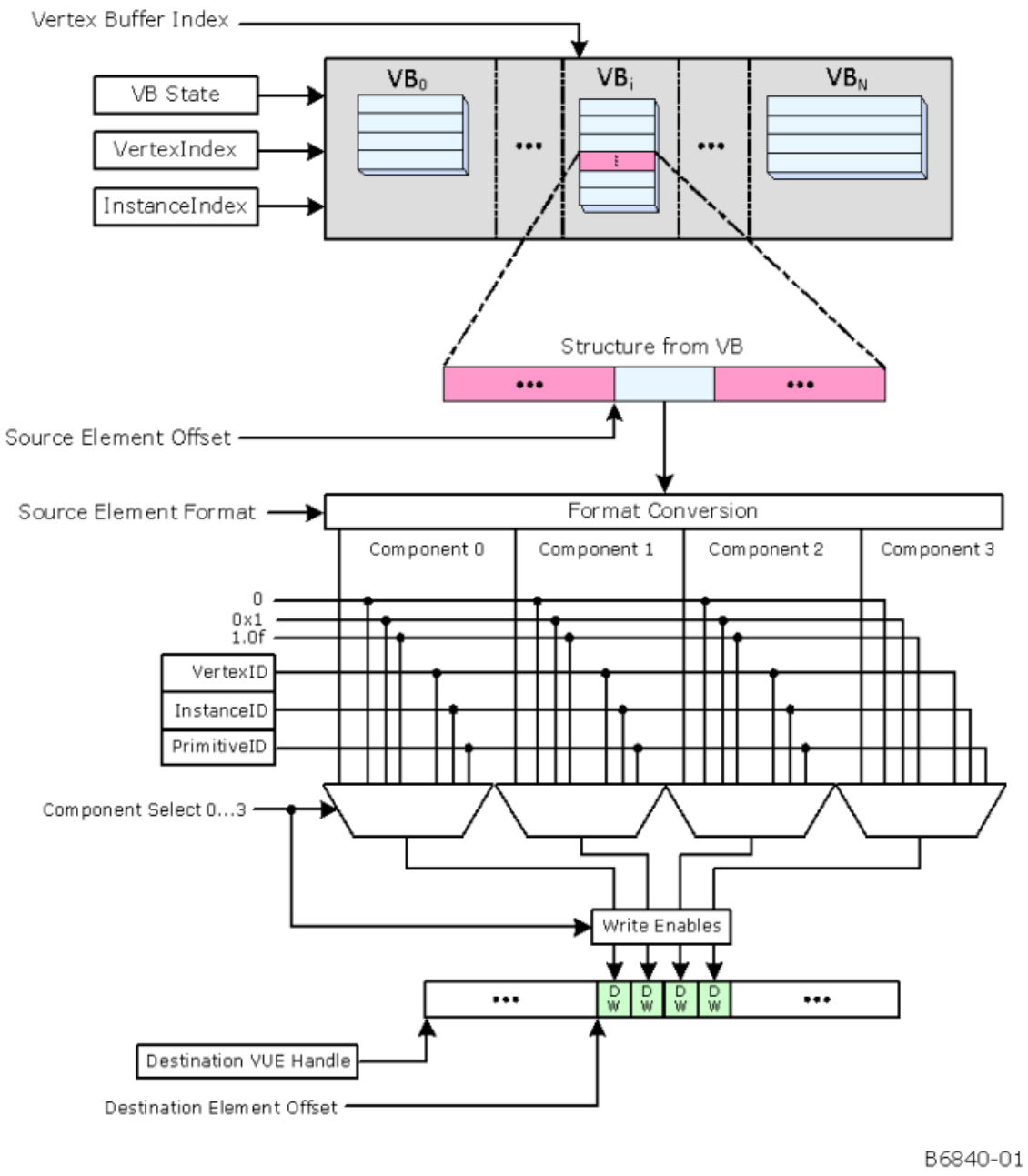
Index Buffer Access

The following figure illustrates how the Index Buffer is accessed.



Vertex Element Data Path

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



FormatConversion

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

The following table lists the valid **Source Element Format** selections, along with the format and availability of the converted components (if a component is listed as -, it cannot be used as the source of a VUE component). **Note:** This table is a subset of the list of supported surface formats defined in the

Sampler chapter. Please refer to that table as the "master list". This table is here only to identify the components available (per format) and their format.

Source Element Formats Supported in VF Unit

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

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
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	-	-	-
R8G8_UNORM	FLOAT	R	G	-	-
R8G8_SNORM	FLOAT	R	G	-	-
R8G8_SINT	SINT	R	G	-	-
R8G8_UINT	UINT	R	G	-	-

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
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	-
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
R10G10B10A2_USCALED	FLOAT	R	G	B	A
R10G10B10A2_SSCALED	FLOAT	R	G	B	A
R10G10B10A2_SINT	SINT	R	G	B	A

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
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 Source Element Formats supported in VF Unit. 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

Payloads

SIMD4x2 Payload

The following table describes the payload delivered to VS threads.

VS Thread Payload (SIMD4x2)

DWord	Bits	Description
	30:0	Reserved
R0.6	31:24	Reserved

DWord	Bits	Description
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Offset:** Specifies the extent 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 *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID:** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. **Format:** Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address**or 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). (See *3D Pipeline* for further description). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)
R0.1	31:16	Reserved

DWord	Bits	Description
	15:0	**URB Return Handle 1:** This is the 64B-aligned URB offset where the EU's upper channels (DWords 7:4) results are to be stored. If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). (See *Generic FF Unit* for further description). Format: **Format** U14 64B-aligned URB offset.
R0.0	31:16	Reserved
	15:0	**URB Return Handle 0:** This is the 64B-aligned URB offset where the EU's lower channels (DWords 3:0) results are to be stored. (See *Generic FF Unit* for further description). Format: **Format** U14 64B-aligned URB offset.
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
Varies	255:0	**Vertex Data:** Data from (possibly) one or (more typically) two Vertex URB Entries is passed to the thread in the thread payload. The **Vertex URB Entry Read Offset** and **Vertex URB Entry Read Length** state variables define the regions of the URB entries that are read from the URB and passed in the thread payload. These SVs can be used to provide a subset of the URB data as required by SW. The vertex data is laid out in the thread header in an interleaved format. The lower DWords (0-3) of these GRF registers always contain data from a Vertex URB Entry. The upper DWords (4-7) may contain data from another Vertex URB Entry. This allows two vertices to be processed (shaded) in parallel SIMD8 fashion. The VS kernel is not aware of the validity of the upper vertex.

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

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. Format: Reserved for HW Implementation Use.
R0.5	31:10	**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 *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID:** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. **Format:** Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = DynamicStateOffset[31:5]
	4	**Description** 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 *3D Pipeline* for further description.) Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2 : R0.0	31:0	Reserved: MBZ

DWord	Bits	Description
R1.7	31:0	Vertex 7 URB Return Handle (see R1.0)
R1.6	31:0	Vertex 6 URB Return Handle (see R1.0)
R1.5	31:0	Vertex 5 URB Return Handle (see R1.0)
R1.4	31:0	Vertex 4 URB Return Handle (see R1.0)
R1.3	31:0	Vertex 3 URB Return Handle (see R1.0)
R1.2	31:0	Vertex 2 URB Return Handle (see R1.0)
R1.1	31:0	Vertex 1 URB Return Handle (see R1.0)
R1.0	31:16	Reserved
	15:0	**Vertex 0 URB Return Handle.** This is the offset within the URB where Vertex 0 is to be stored. Format: 64B-granular offset into the URB
[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 limiit 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

DWord	Bits	Description
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) tessellaton. 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.

Programming Note	
Context:	Hull Shader (HS) Stage
Tessellation is not supported in HPCXTs in which case the HS, TE, and DS stages must all be disabled.	

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.

Programming Note	
Context:	HS Thread Execution - Dual Patch Execution
• In DUAL_PATCH mode HS threads operate on up to two patches, with Patch 0 data residing in the four lower channels and Patch 1 residing in the four upper channels. (Note that, for boundary cases, Patch 1 may not be present, with the upper four channels disabled via Dispatch Mask.) Correspondingly, up to two output patch URB handles are provided. • DUAL_PATCH mode does not support thread instancing,	

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.

DUAL_PATCH Dispatch Mask

In DUAL_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: 0x0000000F
- 2 patches 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 8 DWords of the patch record is defined as a "Patch Header". The 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 following tables show the fixed layouts of the Patch Header DW0-7, depending on DomainType.

Patch Header (QUAD Domain)

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

DWord	Bits	Description
1	31:0	Reserved : MBZ
0	31:1	Reserved : MBZ
	0	Description Reserved: MBZ

Patch Header (TRI Domain)

DWord	Bits	Description
7	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32
6	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
5	31:0	**WEQ0 Tessellation Factor** Format: FLOAT32
4	31:0	**Inside Tessellation Factor** Format: FLOAT32
3-1	31:0	Reserved : MBZ
0	31:1	Reserved : MBZ
	0	Description Reserved: MBZ

Patch Header (ISOLINE Domain)

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

Statistics Gathering

HS Invocations

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

Payloads

SINGLE_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).

SINGLE_PATCH HS Thread Payload

GRF DWord	Bits	Description
R0.7	31	
	30:0	Reserved.
R0.6	31	Dereference Thread This bit is defined to send back the Handle ID back to HS to dereference the input handles for this thread.
	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	Reserved.
	8:0	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 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.

GRF DWord	Bits	Description
R0.3	31:5	Sampler State Pointer. Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved.
	3:0	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 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. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:24	Reserved.
	23:17	Instance Number. A patch-relative instance number between 0 and InstanceCount-1. Format = U7
	16:13	Barrier Index. This index is to be used in any BarrierMsgs sent by this thread to the Gateway. Format = U4
	12:0	Reserved.
R0.1	31:0	Primitive ID. This field contains the Primitive ID associated with the patch. Format: U32
R0.0	31:16	Reserved.
	15:0	Patch Data Record URB Return Handle. Format: **Format** U14 64B-aligned URB offset.
R1 is only included for dispatches that have Include Vertex Handles enabled.		
R1.7	31:16	Reserved.

GRF DWord	Bits	Description
	15:0	ICP 7 Handle Format: Format U14 64B-aligned URB offset.
R1.6	31:16	Reserved.
	15:0	ICP 6 Handle
R1.5	31:16	Reserved.
	15:0	ICP 5 Handle
R1.4	31:16	Reserved.
	15:0	ICP 4 Handle
R1.3	31:16	Reserved.
	15:0	ICP 3 Handle
R1.2	31:16	Reserved.
	15:0	ICP 2 Handle
R1.1	31:16	Reserved.
	15:0	ICP 1 Handle
R1.0	31:16	Reserved.
	15:0	ICP 0 Handle
R2 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count >7		
R2.7	31:16	Reserved.
	15:0	ICP 15 Handle
R2.6	31:16	Reserved.
	15:0	ICP 14 Handle
R2.5	31:16	Reserved.
	15:0	ICP 13 Handle
R2.4	31:16	Reserved.
	15:0	ICP 12 Handle
R2.3	31:16	Reserved.
	15:0	ICP 11 Handle
R2.2	31:16	Reserved.
	15:0	ICP 10 Handle
R2.1	31:16	Reserved.

GRF DWord	Bits	Description
	15:0	ICP 9 Handle
R2.0	31:16	Reserved.
	15:0	ICP 8 Handle
R3 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count >15		
R3.7	31:16	Reserved.
	15:0	ICP 23 Handle
R3.6	31:16	Reserved.
	15:0	ICP 22 Handle
R3.5	31:16	Reserved.
	15:0	ICP 21 Handle
R3.4	31:16	Reserved.
	15:0	ICP 20 Handle
R3.3	31:16	Reserved.
	15:0	ICP 19 Handle
R3.2	31:16	Reserved.
	15:0	ICP 18 Handle
R3.1	31:16	Reserved.
	15:0	ICP 17 Handle
R3.0	31:16	Reserved.
	15:0	ICP 16 Handle
R4 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count >23		
R4.7	31:16	Reserved.
	15:0	ICP 31 Handle
R4.6	31:16	Reserved.
	15:0	ICP 30 Handle
R4.5	31:16	Reserved.
	15:0	ICP 29 Handle
R4.4	31:16	Reserved.
	15:0	ICP 28 Handle
R4.3	31:16	Reserved.
	15:0	ICP 27 Handle
R4.2	31:16	Reserved.
	15:0	ICP 26 Handle

GRF DWord	Bits	Description
R4.1	31:16	Reserved.
	15:0	ICP 25 Handle
R4.0	31:16	Reserved.
	15:0	ICP 24 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] optional	255:0	ICP Vertex Data (optional): There can be up to 32 vertices supplied, each with a size defined by the Vertex URB Entry Read Length state. Vertex 0 DWord 0 is located at Rn.0, Vertex 0 DWord 1 is located at Rn.1, etc. Vertex 1 DWord 0 immediately follows the last DWord of Vertex 0, and so on.

DUAL_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).

HS Thread Payload (DUAL_PATCH Mode)

GRF DWord	Bits	Description
30:0	Reserved	
R0.6	31	Dereference Thread. This bit is defined to send the Handle ID back to HS to dereference the input handles for this thread.
	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 1 KB-aligned offset from the General State Base Address. Format: GeneralStateOffset[31:10]
	9	Reserved
	8:0	FFTID. This ID is assigned by the fixed function unit and is a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use

GRF DWord	Bits	Description
R0.4	31:5	Binding Table Pointer. Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format: SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	Sampler State Pointer. Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format: DynamicStateOffset[31:5]
	4	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). Programming Note: 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. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:24	Reserved
	23:17	Instance Number. A patch-relative instance number between 0 and InstanceCount - 1. Format = U7
	16:13	Barrier Index. This index is used in any BarrierMsgs sent by this thread to the Gateway. Format = U4
	12	Reserved
	11:0	Reserved
R0.1	31:16	Reserved
	15:0	Patch 1 Patch Data Record URB Return Handle. Format: U14 64B-aligned URB Offset
R0.0	31:16	Reserved
	15:0	Patch 0 Patch Data Record URB Return Handle. Format: U14 64B-aligned URB Offset
The following register is included only if Include PrimitiveID is enabled.		
R1.7:R1.5	31:0	Reserved
R1.4	31:0	Primitive ID 1. This field contains the Primitive ID associated with all instances of input Patch 1 (if present). Format: U32
R1.3:R1.1	31:0	Reserved
R1.0	31:0	Primitive ID 1. This field contains the Primitive ID associated with all instances of input Patch 0. Format: U32
The following register is only included for dispatches that have Include Vertex Handles enabled.		

GRF DWord	Bits	Description
31:16	31:16	Reserved
	15:0	Patch 1 ICP 3 Handle Format: U14 64B-aligned URB Offset
R2.6	31:16	Reserved
	15:0	Patch 1 ICP 2 Handle
R2.5	31:16	Reserved
	15:0	Patch 1 ICP 1 Handle
R2.4	31:16	Reserved
	15:0	Patch 1 ICP 0 Handle
R2.3	31:16	Reserved
	15:0	Patch 0 ICP 3 Handle
R2.2	31:16	Reserved
	15:0	Patch 0 ICP 2 Handle
R2.1	31:16	Reserved
	15:0	Patch 0 ICP 1 Handle
R2.0	31:16	Reserved
	15:0	Patch 0 ICP 0 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] optional.	255:0	ICP Vertex Data (optional). There can be up to 4 ICPs (vertices) supplied per patch, each with a size defined by the Vertex URB Entry Read Length state. The pushed vertex data is split into upper and lower halves with Patch 0 input vertices in the lower half, and Patch 1 input vertices in the upper half. The pushed data for Patch 0,1 Vertex 0 immediately follows any pushed constant data. The pushed data for Patch 0,1 Vertex 1 immediately follows Vertex 0, and so on.

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	Reserved
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the General State Base Address. Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID.** This ID is assigned by the fixed function unit and is a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	**Reserved**
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	**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	31:24	**Reserved**

GRF DWord	Bits	Description
	23:17	**Instance Number.** A patch-relative instance number between 0 and InstanceCount-1. Format = U4
	16:13	**Barrier Index.** This index is to be used in any BarrierMsgs sent by this thread to the Gateway. Format = U4
	12:0	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)
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

GRF DWord	Bits	Description
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
	15:0	**Patch 6 ICP 0 Handle**
Rn.5	31:16	Reserved
	15:0	**Patch 5 ICP 0 Handle**
Rn.4	31:16	Reserved
	15:0	**Patch 4 ICP 0 Handle**
Rn.3	31:16	Reserved
	15:0	**Patch 3 ICP 0 Handle**
Rn.2	31:16	Reserved
	15:0	**Patch 2 ICP 0 Handle**
Rn.1	31:16	Reserved
	15:0	**Patch 1 ICP 0 Handle**
Rn.0	31:16	Reserved
	15:0	**Patch 0 ICP 0 Handle**
[Rn+1]	255:0	**ICP 1 Handle for Patches 0-7**
[Rn+2]	255:0	**ICP 2 Handle for Patches 0-7**
...		...
[Rn+31]	255:0	ICP 31 Handle for Patches 0-7
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
Varies		**Pushed Vertex Data (optional)** Input data for the 8 patches is located here. Patch 0 (starting with Vertex 0 of Patch 0) data is

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

Tessellation Engine (TE) Stage

When enabled, the Tessellation Engine (TE) stage performs fixed-function domain tessellation (decomposition into smaller objects) of incoming patches, as referenced by an HS-generated input PDR handle and as controlled by TE state and Tessellation Factors (TFs) read from the Patch URB Entry (patch record). The TE stage is entirely fixed-function and does not spawn threads.

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 Tesellation 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.

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

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

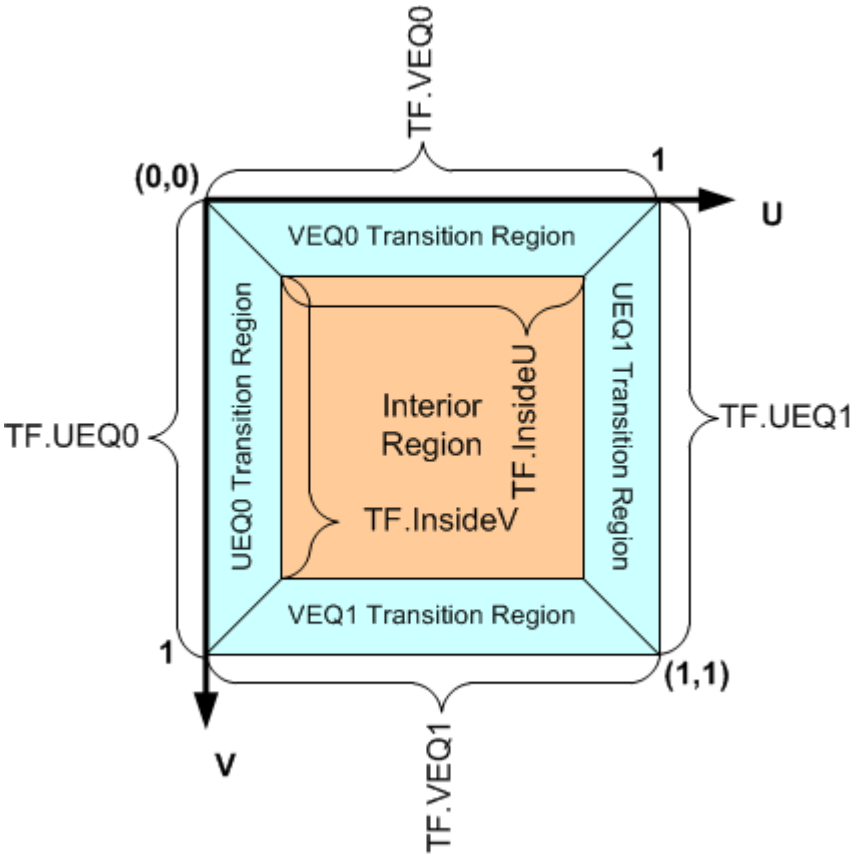
QUAD Domain Tessellation

The four "outside" TFs (TF.UEQ0, TF.VEQ0, TF.UEQ1, TF.VEQ1) are used to specify the level of tessellation along the four corresponding edges of the 2D quad domain. The two "inside" TFs (TF.InsideU, TF.InsideV)

are used to determine the level of tessellation within a 2D “interior” region. Typically the interior region appears as a “regularly-tessellated 2D grid”, however under certain conditions the interior region may collapse in which case only the outside TFs are relevant.

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

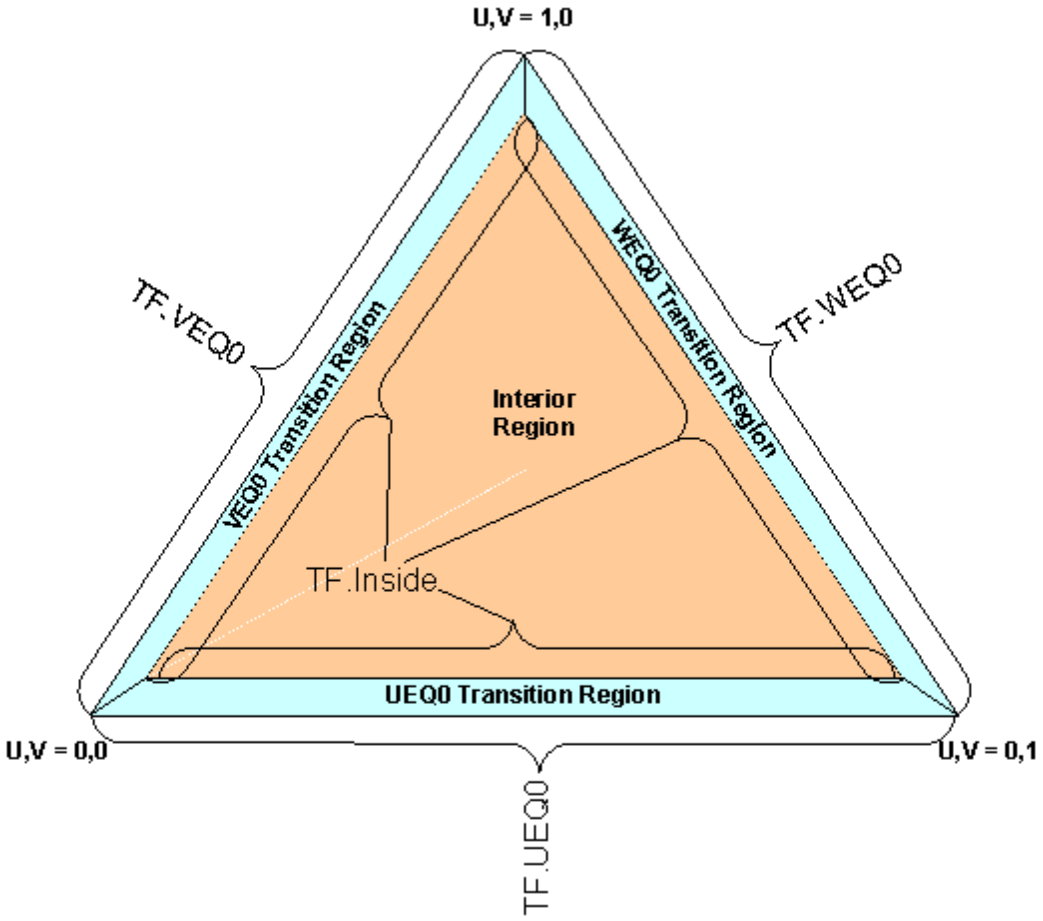
QUAD Domain



TRI Domain Tessellation

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

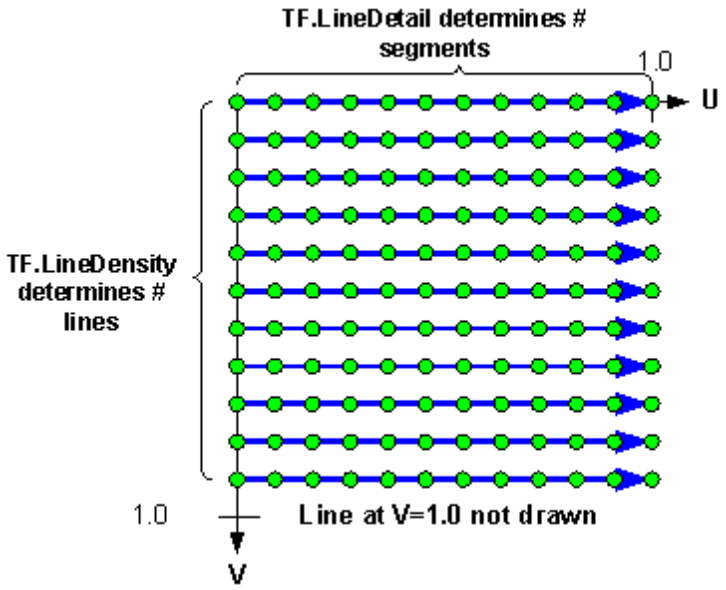
TRI Domain



ISOLINE Domain Tessellation

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

ISOLINE Domain



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.

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

This topic is currently under development.

SIMD4x2 Thread Execution

Description
A DS kernel assumes it is to operate on up to eight domain points in parallel using the EU's SIMD8 execution model, or on two domain points in parallel (using the EU's SIMD4x2 execution model).

Refer to the ISA chapters for specifics on writing kernels that operate in SIMD4x2 fashion.

DS threads must always write the destination URB handles passed in the payload. DS threads are not permitted to request additional destination handles. Refer to 3D Pipeline Stage Overview (*3D Overview*) for details on how destination vertices are written and any required contents/formats.

DS threads must signal thread termination on the last message output to the URB shared function.

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

This topic is currently under development.

SIMD4x2 Payload

The following table describes the payload delivered to DS threads.

DS Thread Payload (SIMD4x2)

DWord	Bits	Description
R0.7	31	
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space 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 *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID.** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use. 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]
	4:0	Reserved

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>R0.3</td><td>31:5</td><td>Sampler State Pointer. Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address.
Format = DynamicStateOffset[31:5]</td></tr>
<tr><td></td><td>4</td><td>Reserved</td></tr>
<tr><td></td><td>3:0</td><td>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).
Format = U4 power of two (in excess of 10)
Range = [0,11] indicating [1K Bytes, 2M Bytes]</td></tr>
<tr><td>R0.2</td><td>31:0</td><td>Reserved: delivered as zeros (reserved for message header fields)</td></tr>
<tr><td>R0.1</td><td>31:26</td><td>Reserved</td></tr>
<tr><td></td><td>25:16</td><td><table><tr><th>Description</th></tr><tr><td>Reserved</td></tr></table></td></tr>
<tr><td></td><td>15:14</td><td>Reserved</td></tr>
<tr><td></td><td>13:0</td><td>URB Return Handle 1: This is the URB handle where Vertex 1 data (the EU’s upper channels (DWords 7:4)) results are to be stored.
If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread’s Channel Mask).
Format:<table><tr><th>Format</th></tr><tr><td>U14 handle (512-bit granular)</td></tr></table></td></tr>
<tr><td>R0.0</td><td>31:26</td><td>Reserved</td></tr>
<tr><td></td><td>25:16</td><td><table><tr><th>Description</th></tr><tr><td>Reserved</td></tr></table></td></tr>
<tr><td></td><td>15:14</td><td>Reserved</td></tr>
<tr><td></td><td>13:0</td><td>URB Return Handle 0: This is the URB handle where Vertex 0 data (the EU’s lower channels (DWords 3:0)) results are to be stored.
Format:<table><tr><th>Format</th></tr><tr><td>U14 handle (512-bit granular)</td></tr></table></td></tr>
</table>

DWord	Bits	Description
R1.7	31:0	**PrimitiveID.** This is the 32-bit PrimitiveID value associated with the patch. It is common to all output vertices resulting from the tessellation of the patch. Format: U32
R1.6	31:0	**Domain Point 1 W Coordinate.** (See Domain Point 0 W Coordinate) Format: FLOAT32
R1.5	31:0	**Domain Point 1 V Coordinate.** (See Domain Point 0 V Coordinate) Format: FLOAT32
R1.4	31:0	**Domain Point 1 U Coordinate.** (See Domain Point 0 U Coordinate) Format: FLOAT32
R1.3	31:14	Reserved
	13:0	**Patch URB Handle.** This is the URB handle of the Patch Record (common to both vertices). Format: Format U14 handle)
R1.2	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
R1.1	31:0	**Domain Point 0 V Coordinate.** V coordinate associated with Domain Point 0. Format: FLOAT32
R1.0	31:0	**Domain Point 0 U Coordinate.** U coordinate associated with Domain Point 0. Format: FLOAT32
Varies [Optional]	255:0	**Constant Data (optional).** Some amount of constant data (possible none) can be extracted from the push constant buffer (PCB) and passed to the thread following the R0 Header. The amount of data provided is defined by the sum of the read lengths in the last 3DSTATE_CONSTANT_DS command (taking the buffer enables into account). The Constant Data arrives in a non-interleaved format.

DWord	Bits	Description
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). The Patch Data arrives in a non-interleaved format.

SIMD8 Payload

The following table describes the payload delivered to DS threads.

DS Thread Payload (SIMD8)

DWord	Bits	Description
R0.7	31	Reserved
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space 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 *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID.** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	Reserved

DWord	Bits	Description
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	**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). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)
R0.1	31:0	**PrimitiveID.** This is the 32-bit PrimitiveID value associated with the patch. It is common to all output domain points resulting from the tessellation of the patch. Format: U32
R0.0	31:27	Reserved
	26:16	**Description** Reserved
	15:0	**Patch URB Offset.** This is the offset within the URB where the patch data is stored. Format: U14 64B-granular offset into the URB
R1.7	31:0	**Domain Point 7 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.6	31:0	**Domain Point 6 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.5	31:0	**Domain Point 5 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.4	31:0	**Domain Point 4 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.3	31:0	**Domain Point 3 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.2	31:0	**Domain Point 2 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.1	31:0	**Domain Point 1 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.0	31:0	**Domain Point 0 U Coordinate.** U coordinate associated with Domain Point 0. Format: FLOAT32
R2.7	31:0	**Domain Point 7 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.6	31:0	**Domain Point 6 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.5	31:0	**Domain Point 5 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.4	31:0	**Domain Point 4 V Coordinate.** (See Domain Point 0 V Coordinate.)

DWord	Bits	Description
R2.3	31:0	**Domain Point 3 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.2	31:0	**Domain Point 2 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.1	31:0	**Domain Point 1 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.0	31:0	**Domain Point 0 V Coordinate.** V coordinate associated with Domain Point 0. Format: FLOAT32
R3.7	31:0	**Domain Point 7 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.6	31:0	**Domain Point 6 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.5	31:0	**Domain Point 5 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.4	31:0	**Domain Point 4 W Coordinate.** (See Domain Point 0 W Coordinate.)
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
R0.7	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**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 *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID.** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	**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). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]

DWord	Bits	Description
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)
R0.1	31:27	Reserved
	26:16	Description Reserved
	15:0	**Patch 1 URB Offset.** This is the offset within the URB where the Patch 1 data is stored. Format: U14 64B-granular offset into the URB
R0.0	31:27	Reserved
	26:16	Description Reserved
	15:0	**Patch 0 URB Offset.** This is the offset within the URB where the Patch 0 data is stored. Format: U14 64B-granular offset into the URB
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.)

DWord	Bits	Description
R3.3	31:0	**Patch 0 Domain Point 3 V Coordinate.** (See R3.0.)
R3.2	31:0	**Patch 0 Domain Point 2 V Coordinate.** (See R3.0.)
R3.1	31:0	**Patch 0 Domain Point 1 V Coordinate.** (See R3.0.)
R3.0	31:0	**Patch 0 Domain Point 0 V Coordinate.** V coordinate associated with Domain Point 0. Format: FLOAT32
R4.7	31:0	**Patch 1 Domain Point 3 W Coordinate.** (See R4.0.)
R4.6	31:0	**Patch 1 Domain Point 2 W Coordinate.** (See R4.0.)
R4.5	31:0	**Patch 1 Domain Point 1 W Coordinate.** (See R4.0.)
R4.4	31:0	**Patch 1 Domain Point 0 W Coordinate.** (See R4.0.)
R4.3	31:0	**Patch 0 Domain Point 3 W Coordinate.** (See R4.0.)
R4.2	31:0	**Patch 0 Domain Point 2 W Coordinate.** (See R4.0.)
R4.1	31:0	**Patch 0 Domain Point 1 W Coordinate.** (See R4.0.)
R4.0	31:0	**Patch 0 Domain Point 0 W Coordinate.** If Compute W Coordinate Enable is set, this field will 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: 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.

DWord	Bits	Description
		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 sections contains the state registers for the Geometry Shader.

The state used by GS is defined with this inline state packet.

3DSTATE_GS

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] = (0); [1] = (1); ...; [N-2] = (N-2);	
POINTLIST_BF	N/A	
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices in Payload [<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
LINELIST_ADJ (N is multiple of 4)	[0] = (0,1,2,3); [1] = (4,5,6,7); ...; [(N/4)-1)] = (N-4,N-3,N-2,N-1)	
LINESTRIP (N >= 2)	[0] = (0,1); [1] = (1,2); ...; [N-2] = (N-2,N-1)	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT (N >= 4)	[0] = (0,1,2,3); [1] = (1,2,3,4); ...; [N-4] = (N-4,N-3,N-2,N-1)	LINESTRIP_ADJ_CONT is added for BDW+. 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	Same as LINESTRIP	Handled same as LINESTRIP
LINELOOP (N >= 2)	[0] = (0,1); [1] = (1,2); [N] = (N-1,0);	Not supported after GS.
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST, RECTLIST_SUBPIXEL	Same as TRILIST	Handled same as TRILIST
TRILIST_ADJ (N is multiple of 6)	[0] = (0,1,2,3,4,5); [1] = (6,7,8,9,10,11); ...; [(N/6)-1] = (N-6,N-5,N-4,N-3,N-2,N-1)	

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

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

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

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices in Payload [<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
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.
: PATCHLIST_1 PATCHLIST_2 PATCHLIST_3..32	[0] = (0); [1] = (1); ...; [N-2] = (N-2); [0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1) similar to above	

Thread Execution

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

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

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

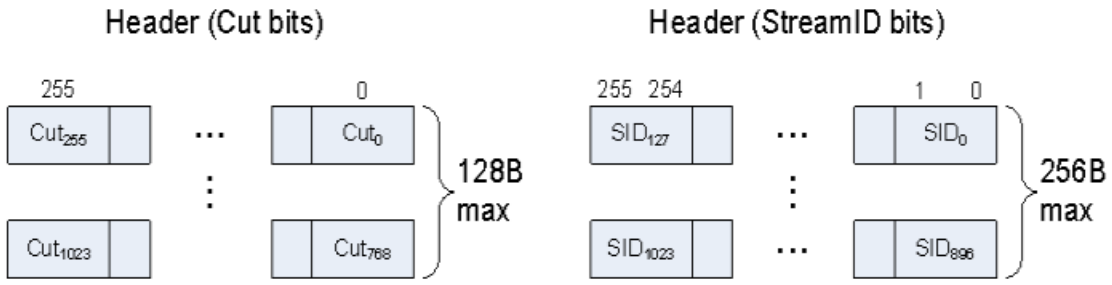
- Emulation of "Polygon Fill Mode", where incoming polygons can be converted to points, lines (wireframe), or solid objects.
- Emulation of wide/sprite points.

When rendering is required, concurrent GS threads must use the FF_SYNC message (URB shared function) to request an initial VUE handle and synchronize output of VUEs to the pipeline (see *URB* in *Shared Functions*). Only one GS thread can be outputting VUEs to the pipeline at a time. To achieve parallelism, GS threads should perform the GS shader algorithm (along with any other required functions) and buffer results (either in the GRF or scratch memory) before issuing the FF_SYNC message. The issuing GS thread is stalled on the FF_SYNC writeback until it is that thread's turn to output VUEs. As only one GS thread at a time can output VUEs, the post-FF_SYNC output portion of the kernel should be optimized as much as possible to maximize parallelism.

Thread Execution

GS URB Entry

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



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

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

room for 1024 scalars / 5 = 204 vertices. 204*64 = 13,056B. Then add 64B to hold 2b/vtx streamID and you get 13,120B entries.

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

GS URB Entry - Output Vertex Count

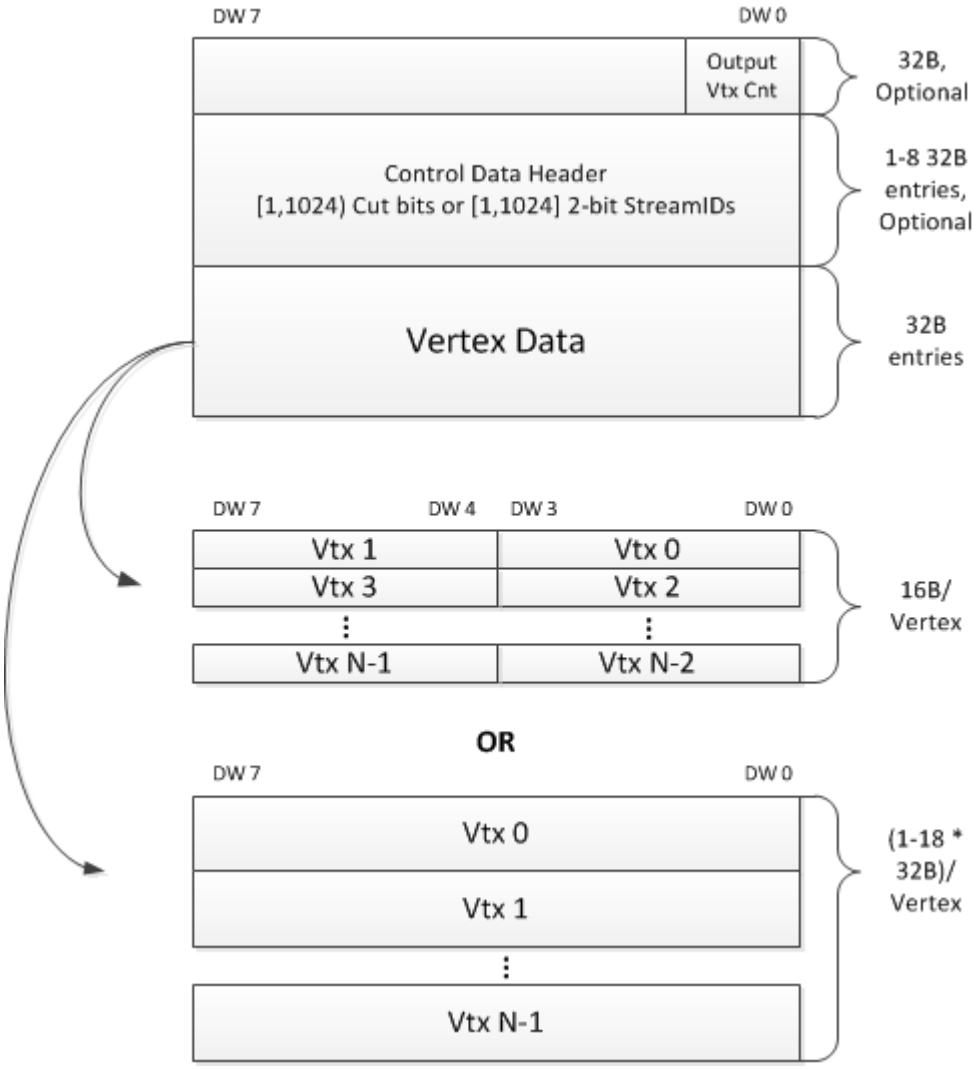
The GS URB entry is the same as in the two previous generations with the following exception: If **Static Output** (3DSTATE_GS) is clear, the URB entry starts with a 32B OUTPUT_VERTEX_COUNT structure as defined below. Ths control header (if present) immediately follows this structure. If **Static Output** 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	Yes
LINESTRIP_CONT	Yes
LINESTRIP_CONT_BF	Yes
LINELOOP	No
POINTLIST	Yes

PrimTopologyType	Supported for GS Thread Output?
POINTLIST_BF	Yes
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 Format** = 0, or
- **Control Data Format** > 0 and **Control Data Format** = GSCTL_CUT

When the GS is enabled, **Control Data Format** > 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/StreamOutput 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).

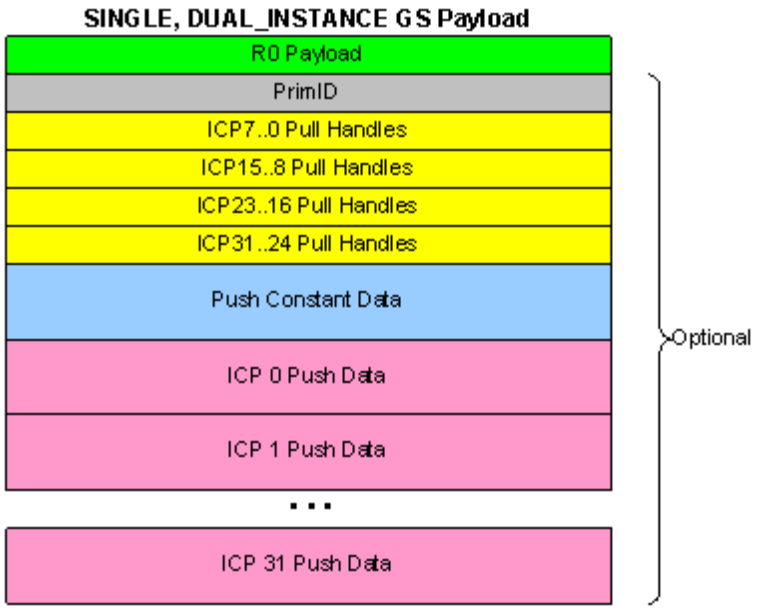
Payloads

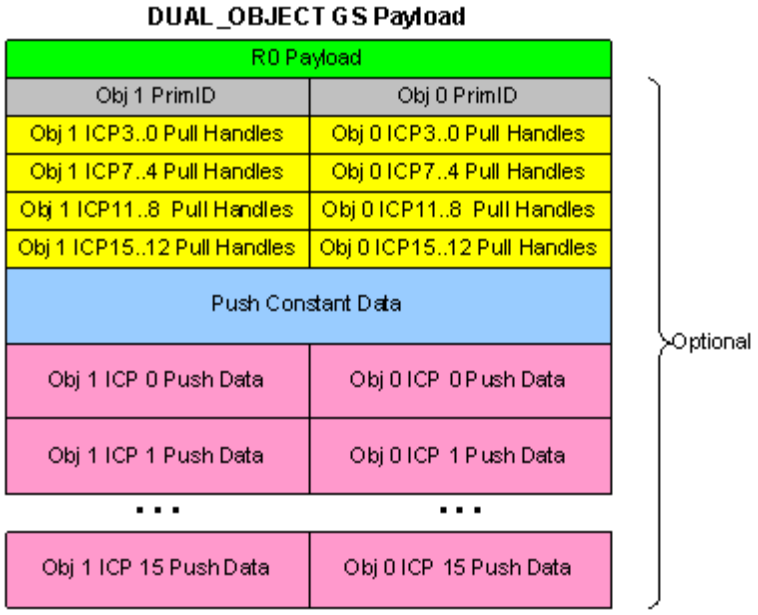
This topic is currently under development.

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 generations.

SIMD 4x2 Thread Payload

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

Refer to 3D Pipeline Stage Overview for details on fields that are common among the various pipeline stages.

GRF DWord	Bits	Description
R0.7	31	Reserved
	30:0	Reserved.
R0.6	31	**Dereference Thread.** This bit is defined to send back the Handle ID back to HS to dereference the input handles for this thread.
	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	Reserved
	8:0	**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.
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 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 for this thread. The value specifies the power that two is raised to (over determine the amount of scratch space). 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. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:24	**Semaphore Index.** This is a DWord index used in URB_ATOMIC commands if the thread is using data pulled from input handles. This information is only required for pull-model vertex inputs and InstanceCount > 1. Format = U8
	23	Reserved.
	22	**Hint.** This is a copy of the corresponding 3DSTATE_GS bit. Format: U1

GRF DWord	Bits	Description
	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*.
	15:13	Reserved
	12:0	**Semaphore Handle.** This is the URB offset pointing to the first GS semaphore DWord in the URB. Software is responsible for statically allocating the semaphore DWords in the URB. Refer to Semaphore Handle field in 3DSTATE_GS for size of semaphore allocation.
R0.1	31:27	**GS Instance ID 1.** 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. If "dispatch mode" is DUAL_OBJECT this field is not valid. Format: U5
	26:16	Reserved.
	15:0	**URB Return Handle 1.** This is the URB offset where the EU's upper channels (DWords 7:4) results are stored. If only one object/instance is processed (shaded) by the thread, this field is effectively ignored (no results are stored for these channels, as controlled by the thread's Channel Mask).
R0.0	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. If "dispatch mode" is DUAL_OBJECT, this field is not valid. 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 stored.
The following register is included only if Include PrimitiveID is enabled.		
R1.7-R1.5	31:0	Reserved: MBZ.
R1.4	31:0	**Primitive ID 1**. This field contains the Primitive ID associated with (all instances) of input object 1. Only valid in DUAL_OBJECT mode. Format: U32
R1.3-R1.1	31:0	Reserved: MBZ.
R1.0	31:0	**Primitive ID 0.** This field contains the Primitive ID associated with (all instances) of input object 0. Format: U32
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled.		
Rn.7	31:16	Description: Reserved
	15:0	ICP 7 Handle

GRF DWord	Bits	Description
Rn.6	31:16	Description: Reserved
	15:0	ICP 6 Handle
Rn.5	31:16	Description: Reserved
	15:0	ICP 5 Handle
Rn.4	31:16	Description: Reserved
	15:0	ICP 4 Handle
Rn.3	31:16	Description: Reserved
	15:0	ICP 3 Handle
Rn.2	31:16	Description: Reserved
	15:0	ICP 2 Handle
Rn.1	31:16	Description: Reserved
	15:0	ICP 1 Handle
Rn.0	31:16	Description: Reserved
	15:0	ICP 0 Handle
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and ICP Count > 7.		
Rn+1.7	31:16	Description: Reserved
	15:0	ICP 15 Handle
Rn+1.6	31:16	Description: Reserved
	15:0	ICP 14 Handle
Rn+1.5	31:16	Description: Reserved
	15:0	ICP 13 Handle

GRF DWord	Bits	Description
Rn+1.4	31:16	Description: Reserved
	15:0	ICP 12 Handle
Rn+1.3	31:16	Description: Reserved
	15:0	ICP 11 Handle
Rn+1.2	31:16	Description: Reserved
	15:0	ICP 10 Handle
Rn+1.1	31:16	Description: Reserved
	15:0	ICP 9 Handle
Rn+1.0	31:16	Description: Reserved
	15:0	ICP 8 Handle
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and ICP Count > 15.		
Rn+2.7	31:16	Description: Reserved
	15:0	ICP 23 Handle
Rn+2.6	31:16	Description: Reserved
	15:0	ICP 22 Handle
Rn+2.5	31:16	Description: Reserved
	15:0	ICP 21 Handle
Rn+2.4	31:16	Description: Reserved
	15:0	ICP 20 Handle
Rn+2.3	31:16	Description: Reserved
	15:0	ICP 19 Handle

GRF DWord	Bits	Description
Rn+2.2	31:16	Description: Reserved
	15:0	ICP 18 Handle
Rn+2.1	31:16	Description: Reserved
	15:0	ICP 17 Handle
Rn+2.0	31:16	Description: Reserved
	15:0	ICP 16 Handle
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and ICP Count > 23.		
Rn+3.7	31:16	Description: Reserved
	15:0	ICP 31 Handle
Rn+3.6	31:16	Description: Reserved
	15:0	ICP 30 Handle
Rn+3.5	31:16	Description: Reserved
	15:0	ICP 29 Handle
Rn+3.4	31:16	Description: Reserved
	15:0	ICP 28 Handle
Rn+3.3	31:16	Description: Reserved
	15:0	ICP 27 Handle
Rn+3.2	31:16	Description: Reserved
	15:0	ICP 26 Handle
Rn+3.1	31:16	Description: Reserved
	15:0	ICP 25 Handle

GRF DWord	Bits	Description
Rn+3.0	31:16	Description: Reserved
	15:0	ICP 24 Handle
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled.		
Rn.7	31:16	Description: Reserved
	15:0	Object 1 ICP 3 Handle
Rn.6	31:16	Description: Reserved
	15:0	Object 1 ICP 2 Handle
Rn.5	31:16	Description: Reserved
	15:0	Object 1 ICP 1 Handle
Rn.4	31:16	Description: Reserved
	15:0	Object 1 ICP 0 Handle
Rn.3	31:16	Description: Reserved
	15:0	Object 0 ICP 3 Handle
Rn.2	31:16	Description: Reserved
	15:0	Object 0 ICP 2 Handle
Rn.1	31:16	Description: Reserved
	15:0	Object 0 ICP 1 Handle
Rn.0	31:16	Description: Reserved
	15:0	Object 0 ICP 0 Handle
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count > 3.		
Rn+1.7	31:16	Description: Reserved
	15:0	Object 1 ICP 7 Handle

GRF DWord	Bits	Description
Rn+1.6	31:16	Description Reserved
	15:0	Object 1 ICP 6 Handle
Rn+1.5	31:16	Description Reserved
	15:0	Object 1 ICP 5 Handle
Rn+1.4	31:16	Description Reserved
	15:0	Object 1 ICP 4 Handle
Rn+1.3	31:16	Description Reserved
	15:0	Object 0 ICP 7 Handle
Rn+1.2	31:16	Description Reserved
	15:0	Object 0 ICP 6 Handle
Rn+1.1	31:16	Description Reserved
	15:0	Object 0 ICP 5 Handle
Rn+1.0	31:16	Description Reserved
	15:0	Object 0 ICP 4 Handle
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count > 7.		
Rn+2.7	31:16	Description Reserved
	15:0	Object 1 ICP 11 Handle
Rn+2.6	31:16	Description Reserved
	15:0	Object 1 ICP 10 Handle
Rn+2.5	31:16	Description Reserved
	15:0	Object 1 ICP 9 Handle

GRF DWord	Bits	Description
Rn+2.4	31:16	Description: Reserved
	15:0	Object 1 ICP 8 Handle
Rn+2.3	31:16	Description: Reserved
	15:0	Object 0 ICP 11 Handle
Rn+2.2	31:16	Description: Reserved
	15:0	Object 0 ICP 10 Handle
Rn+2.1	31:16	Description: Reserved
	15:0	Object 0 ICP 9 Handle
Rn+2.0	31:16	Description: Reserved
	15:0	Object 0 ICP 8 Handle
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count > 11.		
Rn+3.7	31:16	Description: Reserved
	15:0	Object 1 ICP 15 Handle
Rn+3.6	31:16	Description: Reserved
	15:0	Object 1 ICP 14 Handle
Rn+3.5	31:16	Description: Reserved
	15:0	Object 1 ICP 13 Handle
Rn+3.4	31:16	Description: Reserved
	15:0	Object 1 ICP 12 Handle
Rn+3.3	31:16	Description: Reserved
	15:0	Object 0 ICP 15 Handle

GRF DWord	Bits	Description
Rn+3.2	31:16	Description Reserved
	15:0	Object 0 ICP 14 Handle
Rn+3.1	31:16	Description Reserved
	15:0	Object 0 ICP 13 Handle
Rn+3.0	31:16	Description Reserved
	15:0	Object 0 ICP 12 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	31:0	**Pushed Vertex Data.** There can be up to 32 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state. For SINGLE or DUAL_INSTANCE dispatch modes, 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. For DUAL_OBJECT dispatch mode, the pushed vertex data is split into upper and lower halves with Object 0 input vertices in the lower half, and Object 1 input vertices in the upper half.

SIMD8 Thread Payload

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

Refer to the 3D Pipeline Stage Overview section in *3D Pipeline* for details on those fields that are common among the various pipeline stages.

GRF DWord	Bits	Description
R0.7	31	Reserved
	30:0	Reserved.
R0.6	31:24	Reserved.
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the General State Base Address.

GRF DWord	Bits	Description
		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. 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.

GRF DWord	Bits	Description
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: U14 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) 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

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

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

GRF DWord	Bits	Description
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.

The SOL unit incorrectly double buffers MMIO/NP registers and only moves them into the design for usage when control topology is received with the SOL unit dirty state.

If the state does not change, need to resend the same state.

There is no need to send a pipeline state update to the SOL unit after SOL unit MMIO registers or non-pipeline state are written.

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

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

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

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

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

Functions

Input Buffering

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

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

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

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

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

PrimTopologyType	Order of Vertices Streamed Out	
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	Any SOL Notes
POINTLIST POINTLIST_BF	[0] = (0); [1] = (1); ...; [N-2] = (N-2);	
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	
LINELIST_ADJ (N is multiple of 4)	[0] = (1,2); [1] = (5,6); ...; [(N/4)-1)] = (N-3,N-2)	

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
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 for BDW+. 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.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
TRISTRIP_REV (N >= 3) REORDER_LEADING	[0] = (0,2,1) [1] = (1,2,3);...; [k even] = (k,k+2,k+1) [k odd] = (k,k+1,k+2) [N-3] = (see above)	"Even" triangles have vertices reordered to yield increasing leading vertices starting with v0.
TRISTRIP_REV (N >= 3) REORDER_TRAILING	[0] = (1,0,2) [1] = (1,2,3);...; [k even] = (k+1,k,k+2) [k odd] = (k,k+1,k+2) [N-3] = (see above)	"Even" triangles have vertices reordered to yield increasing trailing vertices starting with v2.
TRISTRIP_ADJ (N even, N >= 6) REORDER_LEADING	N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (2,6,4); ...; N > 10: [0] = (0,2,4); [1] = (2,6,4); ...; [k>1, even] = (2k, 2k+2, 2k+4); [k>2, odd] = (2k, 2k+4, 2k+2);...; Trailing object: [(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,	"Odd" objects have vertices reordered to yield increasing-by-2 trailing vertices starting with v4.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
	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);	
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); ...; [N-3] = (0, N-2, N-1);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is added for BDW+. POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
QUADLIST QUADSTRIP	N/A	Not supported after VF.
PATCHLIST_1	[0] = (0); [1] = (1); ...; [N-2] = (N-2);	
PATCHLIST_2	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	
PATCHLIST_3..32	similar to above	

Stream Output Function

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

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

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

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

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

Surface Format Name
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.

3DSTATE_VIEWPORT_STATE_POINTERS_CC

3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP

3DSTATE_SCISSOR_STATE_POINTERS

3DSTATE_RASTER

3D Pipeline – CLIP Stage Overview

The CLIP stage of the GEN 3D Pipeline is similar to the GS stage in that it can be used to perform general processing on incoming 3D objects via spawned GEN4 threads. However, the CLIP stage also includes specialized logic to perform a *ClipTest* function on incoming objects. These two usage models of the CLIP stage are outlined below.

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

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

Clip Stage – 3D Clipping

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

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

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

Fixed Function Clipper

The GPU supports Fixed Function Clipping.

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

Concepts

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

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

CLIP Stage Input

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

State

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

3DSTATE_CLIP

3D_STATE_CLIP

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

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

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

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

VUE Readback

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

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

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

VertexClipTest Function

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

The RHW component of the incoming vertex position is tested for NaN value, and if a NaN is detected, the vertex is marked as "BAD" by setting the outcode[BAD]. If a NaN is detected in any vertex homogeneous x,y,z,w component or an enabled ClipDistance value, the vertex is marked as "BAD" by setting the outcode[BAD].

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

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

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

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

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

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

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

The following algorithm is used by VertexClipTest:

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

```
//
// If 1/w is negative, w is negative and therefore outside of the w=0 plane
//
//
rhw_neg = ISNEG(rhw)
if (rhw_neg)
{
   outCode[NEGW] = 1
}
//
// View Volume Clip Test
// If Premapped, the 2D viewport is defined in screen space
// otherwise the canonical NDC viewvolume applies ([-1,1])
//
if (CLIP_STATE.PreMapped)
{
   vp_XMIN = CLIP_STATE.VP_XMIN
   vp_XMAX = CLIP_STATE.VP_XMAX
   vp_YMIN = CLIP_STATE.VP_YMIN
   vp_YMAX = CLIP_STATE.VP_YMAX
} else {
   vp_XMIN = -1.0f
   vp_XMAX = +1.0f
   vp_YMIN = -1.0f
   vp_YMAX = +1.0f
}

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

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

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

   #ifdef (BW-E0)
   if (rhw_neg) {
```

```
      outCode[VP_ZMIN] = (z >= vp_ZMIN)
      outCode[VP_ZMAX] = (z <= vp_ZMAX)
   }
   #endif
   if (rhw_neg) {
      outCode[VP_ZMIN] = (z > vp_ZMIN)
      outCode[VP_ZMAX] = (z < vp_ZMAX)
   }
}
//
// Guardband Clip Test
//
if {CLIP_STATE.GuardbandClipTestEnable) {
   gb_XMIN = CLIP_STATE.Viewport[vpindex].GB_XMIN
   gb_XMAX = CLIP_STATE.Viewport[vpindex].GB_XMAX
   gb_YMIN = CLIP_STATE.Viewport[vpindex].GB_YMIN
   gb_YMAX = CLIP_STATE.Viewport[vpindex].GB_YMAX
   outCode[GB_XMIN] = (x < gb_XMIN)
   outCode[GB_XMAX] = (x > gb_XMAX)
   outCode[GB_YMIN] = (y < gb_YMIN)
   outCode[GB_YMAX] = (y > gb_YMAX)

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

```
if (isNAN(y)) {
   outCode[GB_YMIN] = xyorgben
   outCode[GB_YMAX] = xyorgben
   outCode[VP_YMIN] = xyorgben
   outCode[VP_YMAX] = xyorgben
}
if (isNaN) {
   outCode[VP_ZMIN] = CLIP_STATE.ViewportZClipTestEnable
   outCode[VP_ZMAX] = CLIP_STATE.ViewportZClipTestEnable
}
//
// UserClipFlags
//
ExamineUCFs
for (i=0; i<7; i++)
{
   outCode[UC0+i] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[i]
}
outCode[UC7] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
```

Object Staging

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

Partial Object Removal

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

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

ClipDetermination Function

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

TR/TA against interesting boundary subsets are then computed. The TR status is computed as the logical OR of the appropriate objANDCode bits, as the vertices need only be outside of one common boundary

to be trivially rejected. The TA status is computed as the logical NOR of the appropriate objORCode bits, as any vertex being outside of any of the boundaries prevents the object from being trivially accepted.

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

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

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way, objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

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

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

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

//
```

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

//
// Trivial Accept
//
// For an object to be TA, it must be TA against the VPZ and GB, not TR,
// and considered TA against the UC planes and NEGW
// If the UCMC mode is disabled, an object is considered TA against the UC
// as long as it isn't TR against the UC.
// If the UCMC mode is enabled, then the object really has to be TA against the UC
// to be considered TA
// In this way, enabling the UCMC mode will force clipping if the object is neither
// TA or TR against the UC
//
TA   = !TR && TA_GB && TA_VPZ && TA_NEGW
UCMC = CLIP_STATE.UserClipFlagsMustClipEnable
TA   = TA && ( (UCMC && TA_UC) || (!UCMC && !TR_UC) )

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

ClipMode State

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

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

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

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

```
            DISCARD  = 0
            SPAWN    = 1
         }
      case CLIP_NOT_REJECT:
         {
            PASSTHRU = 0
            DISCARD  = TR   || BAD
            SPAWN    = !(TR || BAD)
         }
      case REJECT_ALL:
         {
            PASSTHRU = 0
            DISCARD  = 1
            SPAWN    = 0
         }
      case ACCEPT_ALL:
         {
            PASSTHRU = !BAD
            DISCARD  = BAD
            SPAWN    = 0
         }
   } endswitch
```

NORMAL ClipMode

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

CLIP_ALL ClipMode

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

CLIP_NON_REJECT ClipMode

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

REJECT_ALL ClipMode

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

ACCEPT_ALL ClipMode

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

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

Object Pass-Through

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

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

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
POINTLIST	POINTLIST	
POINTLIST_BF	POINTLIST_BF	
LINELIST	LINELIST	
LINELIST_ADJ	LINELIST	Adjacent vertices removed.
LINESTRIP	LINESTRIP	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT	LINESTRIP	Adjacent vertices removed. LINESTRIP_ADJ_CONT is added for BDW+. LINESTRIP_ADJ_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
LINESTRIP_BF	LINESTRIP_BF	
LINESTRIP_CONT	LINESTRIP_CONT	
LINESTRIP_CONT_BF	LINESTRIP_CONT_BF	
LINELOOP	N/A	Not supported after GS.
TRILIST	TRILIST	
RECTLIST	RECTLIST	
TRILIST_ADJ	TRILIST	Adjacent vertices removed.

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
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 for BDW+. 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

The CLIP unit includes logic to assist in the gathering of certain pipeline statistics . The statistics take the form of MI counter registers (see Memory Interface Registers), where the CLIP unit provides signals causing those counters to increment.

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

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

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

CL_INVOCATION_COUNT

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

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

3D Pipeline - Strips and Fans (SF) Stage

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

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

Inputs from CLIP

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

SF's Vertex Pipeline Inputs

Variable	Type	Description
primType	enum	Type of primitive topology the vertex belongs to. *Primitive Assembly* for a list of primitive types supported by the SF unit. See *3D Pipeline* for descriptions of these topologies. Notes: The CLIP unit will convert any primitive with adjacency (3DPRIMxxx_ADJ) it receives from the pipeline into the corresponding primitive without adjacency (3DPRIMxxx). QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit. Software must use a GS thread to convert these to some other (supported) primitive type.

Variable	Type	Description
		If an object is clipped by the hardware clipper, the CLunit would force this field to "3DPRIM_POLYGON". SFunit would process this incoming object just as it would any other "3DPRIM_POLYGON". SFunit selects vertex 0 as the provoking vertex.
primStart,primEnd	boolean	Indicate vertex's position within the primitive topology
vInX[]	float	Vertex X position (screen space or NDC space)
vInY[]	float	Vertex Y position (screen space or NDC space)
vInZ[]	float	Vertex Z position (screen space or NDC space)
vInInvW[]	float	Reciprocal of Vertex homogeneous (clip space) W
hVUE[]	URB address	Points to the vertex's data stored in the URB (one VUE handle per vertex)
renderTargetArrayIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in subsequent processing.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in the Viewport Transform and Scissor subfunctions, otherwise the value is ignored. Note that for primitive topologies with vertices shared between objects, this means a shared vertex may be subject to multiple Viewport Transformation operations if the viewPortIndex varies within the topology.
pointSize	uint	If this vertex is within a POINTLIST[_BF] primitive topology, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise the value is ignored.

Attribute Setup/Interpolation Process

The following sections describe the Attribute Setup/Interpolation Process.

Attribute Setup/Interpolation Process

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

Outputs to WM

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

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

Primitive Assembly

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

The number of vertices passed with each primitive is constrained by the primitive type. *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	>=2

primType	VertexCount Restriction
3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	
3DPRIM_RECTLIST	nonzero multiple of 3
3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	nonzero

Primitive Assembly 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 Assembly for the outputs of Primitive Decomposition.

Primitive Decomposition Outputs

Variable	Type	Description
objectType	enum	Type of object. *Primitive Assembly*
nV	uint	The number of object vertices passed to Object Setup. *Primitive Assembly*
v[0..nV-1]*	various	Data arrays associated with object vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. *Primitive Assembly*

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

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

VPIndex/RTAIndex Selection

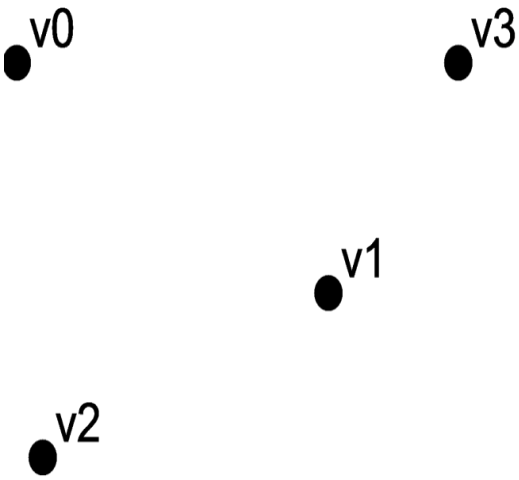
PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex→ Line(V0,V1) V2.VPIndex→ Line(V2,V3) ...
LINESTRIP LINESTRIP_BF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

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

Point List Decomposition

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

3DPRIM_POINTLIST Primitive



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

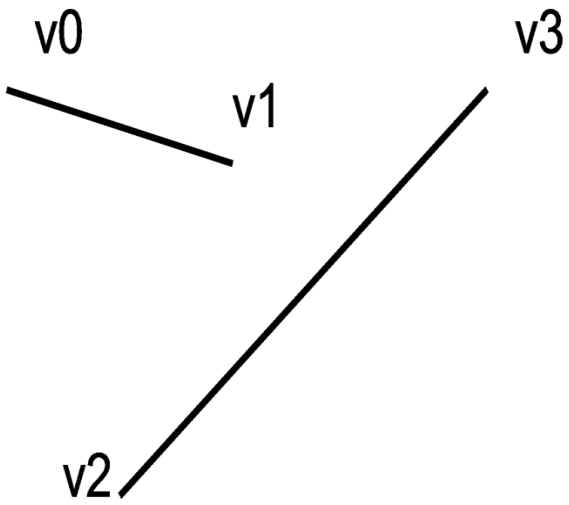
Points have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing points. Primitives of type 3DPRIM_POINTLIST_BACKFACING are decomposed exactly the same way as 3DPRIM_POINTLIST primitives, but the *backFacing* variable is set for resulting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

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

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

Line List Decomposition

The 3DPRIM_LINELIST primitive specifies a list of independent lines.

3DPRIM_LINELIST Primitive



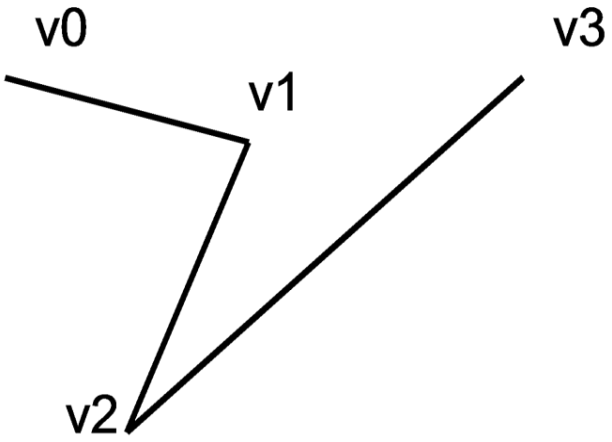
The decomposition process divides the list into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0, v1; v2, v3; and so on. The *provokingVertex* of each object is taken from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

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

Line Strip Decomposition

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

3DPRIM_LINESTRIP_xxx Primitive



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

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

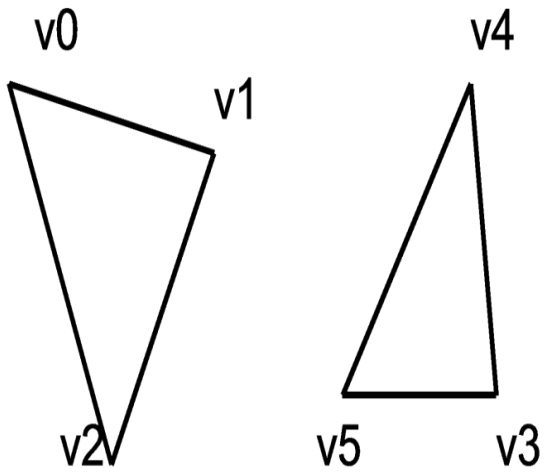
```
LineStripDecomposition()
   {
      objectType = 3DOBJ_LINE
      nV = 2
      provokingVtx = Line List/Strip Provoking Vertex Select
      if (primType == 3DPRIM_LINESTRIP)
         {
             backFacing = FALSE
             continueStipple = FALSE
         } else if (primType == 3DPRIM_LINESTRIP_BF)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

```
            v[1] arrays ← vIn[i+1] arrays
            ObjectSetup()
            continueStipple = TRUE
        }
    }
```

Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

3DPRIM_TRILIST Primitive



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

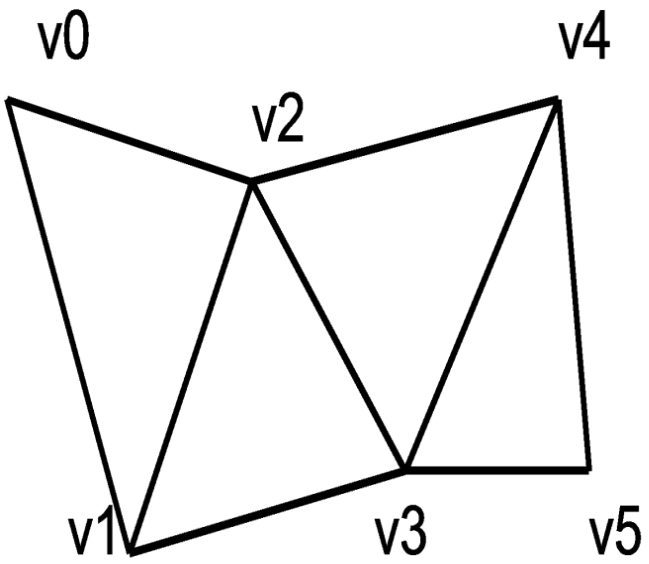
```
TriangleListDecomposition() {
  objectType = 3DOBJ_TRIANGLE
      nV = 3
  invertOrientation = FALSE
  provokingVtx = Triangle List/Strip Provoking Vertex Select
  polyStippleEnable = TRUE
  for each (vertex I in [0..vertexCount-3] by 3)
      {

          v[0] arrays ⟵ vIn[i] arrays


          v[1] arrays ⟵ vIn[i+1] arrays


          v[2] arrays ⟵ vIn[i+2] arrays

          ObjectSetup()
```

```
        }
}
```

Triangle Strip Decomposition

The 3DPRIM_TRISTRIP and 3DPRIM_TRISTRIP_REVERSE primitives specify a series of triangles arranged in a strip, as illustrated below.

3DPRIM_TRISTRIP[_REVERSE] Primitive



The decomposition process divides the strip into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v1,v2,v3; v2,v3,v4; and so on. Note that the *winding order* of the vertices alternates between CW (clockwise), CCW (counter-clockwise), CW, etc. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

The 3D pipeline uses the winding order of the vertices to distinguish between front-facing and back-facing triangles (*Triangle Orientation (Face) Culling* below). Therefore, the 3D pipeline must account for the alternation of winding order in strip triangles. The *invertOrientation* variable is generated and used for this purpose.

To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (for example, due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

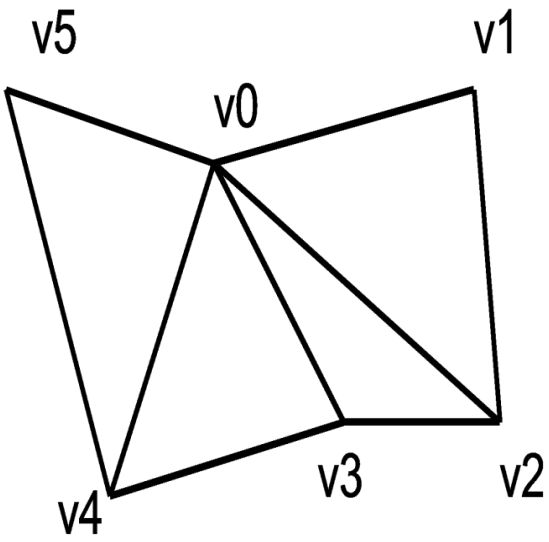
```
TriangleStripDecomposition()
   {
```

```
        objectType = 3DOBJ_TRIANGLE
        nV = 3
        provokingVtx = Triangle List/Strip Provoking Vertex Select
        if (primType == 3DPRIM_TRISTRIP)
           invertOrientation = FALSE
           else // primType == 3DPRIM_TRISTRIP_REVERSE
              invertOrientation = TRUE
        polyStippleEnable = TRUE
        for each (vertex I in [0..vertexCount-3])
          {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              v[2] arrays ← vIn[i+2] arrays
              ObjectSetup()
              invertOrientation = ! invertOrientation
          }
    }
```

Triangle Fan Decomposition

The 3DPRIM_TRIFAN and 3DPRIM_TRIFAN_NOSTIPPLE primitives specify a series of triangles arranged in a fan, as illustrated below.

3DPRIM_TRIFAN Primitive



The decomposition process divides the fan into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v0,v2,v3; v0,v3,v4; and so on. As there is no alternation in the vertex winding order, the *invertOrientation* variable is output as FALSE unconditionally. The *provokingVertex* of each object is taken from the **Triangle Fan Provoking Vertex** state variable, as programmed via SF_STATE.

Primitives of type 3DPRIM_TRIFAN_NOSTIPPLE are decomposed exactly the same way, except the *polyStippleEnable* variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```
TriangleFanDecomposition()
   {
       objectType = 3DOBJ_TRIANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = Triangle Fan Provoking Vertex Select
       if (primType == 3DPRIM_TRIFAN)
           polyStippleEnable = TRUE
           else // primType == 3DPRIM_TRIFAN_NOSTIPPLE
              polyStippleEnable = FALSE

       v[0] arrays ← vIn[0] arrays
         // the 1st vertex is common

       for each (vertex I in [1..vertexCount-2])
          {
              v[1] arrays ← vIn[i] arrays
              v[2] arrays ← vIn[i+1] arrays
              ObjectSetup()
          }
   }
```

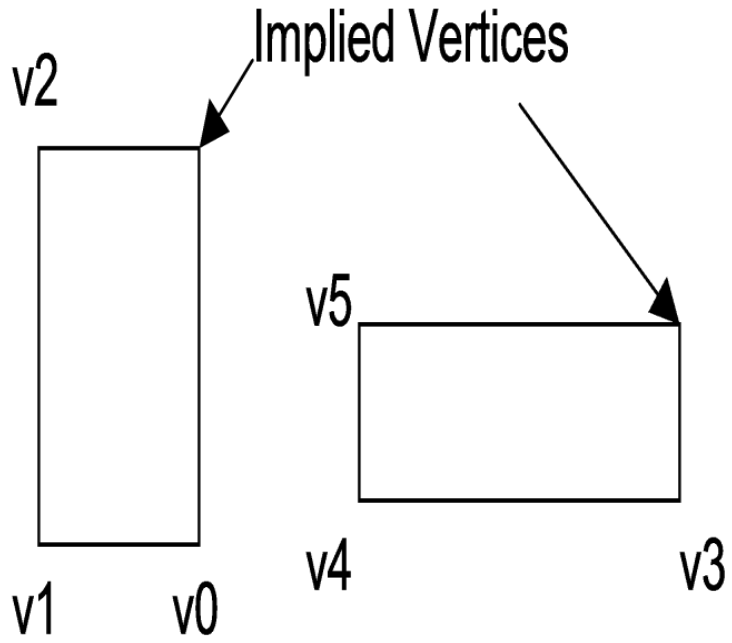
Polygon Decomposition

The 3DPRIM_POLYGON primitive is identical to the 3DPRIM_TRIFAN primitive with the exception that the *provokingVtx* is overridden with 0. This support has been added specifically for OpenGL support, avoiding the need for the driver to change the provoking vertex selection when switching between trifan and polygon primitives.

Rectangle List Decomposition

The 3DPRIM_RECTLIST primitive command specifies a list of independent, axis-aligned rectangles. Only the lower right, lower left, and upper left vertices (in that order) are included in the command – the upper right vertex is derived from the other vertices (in Object Setup).

3DPRIM_RECTLIST Primitive



The decomposition of the 3DPRIM_RECTLIST primitive is identical to the 3DPRIM_TRILIST decomposition, with the exception of the *objectType* variable.

```
RectangleListDecomposition()
   {
       objectType = 3DOBJ_RECTANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = 0
       for each (vertex I in [0..vertexCount-3] by 3)
          {
              v[0] arrays ←  vIn[i] arrays
              v[1] arrays ←  vIn[i+1] arrays
              v[2] arrays ←  vIn[i+2] arrays
              ObjectSetup()
          }
   }
```

Object Setup

The Object Setup subfunction of the SF stage takes the post-viewport-transform data associated with each vertex of a basic object and computes various parameters required for scan conversion. This includes generation of implied vertices, translations and adjustments on vertex positions, and culling (removal) of certain classes of objects. The final object information is passed to the Windower/Masker (WM) stage where the object is rasterized into pixels.

Invalid Position Culling (Pre/Post-Transform)

At input the the SF stage, any objects containing a floating-point NaN value for Position X, Y, Z, or RHW will be unconditionally discarded. Note that this occurs on an object (not primitive) basis.

If Viewport Transformation is enabled, any objects containing a floating-point NaN value for post-transform Position X, Y or Z will be unconditionally discarded.

Viewport Transformation

If the **Viewport Transform Enable** bit of SF_STATE is ENABLED, a viewport transformation is applied to each vertex of the object.

The VPIndex associated with the leading vertex of the object is used to obtain the **Viewport Matrix Element** data from the corresponding element of the SF_VIEWPORT structure in memory. For each object vertex, the following scale and translate transformation is applied to the position coordinates:

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

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

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

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

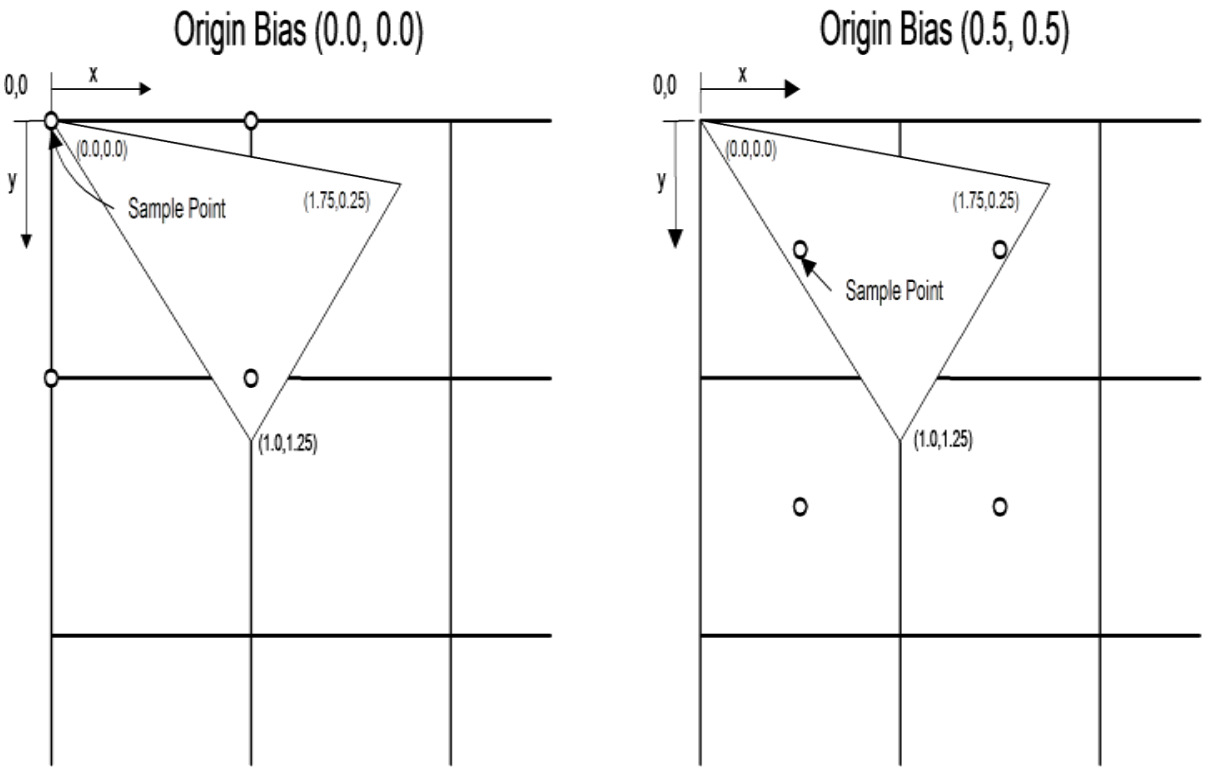
Destination Origin Bias

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.

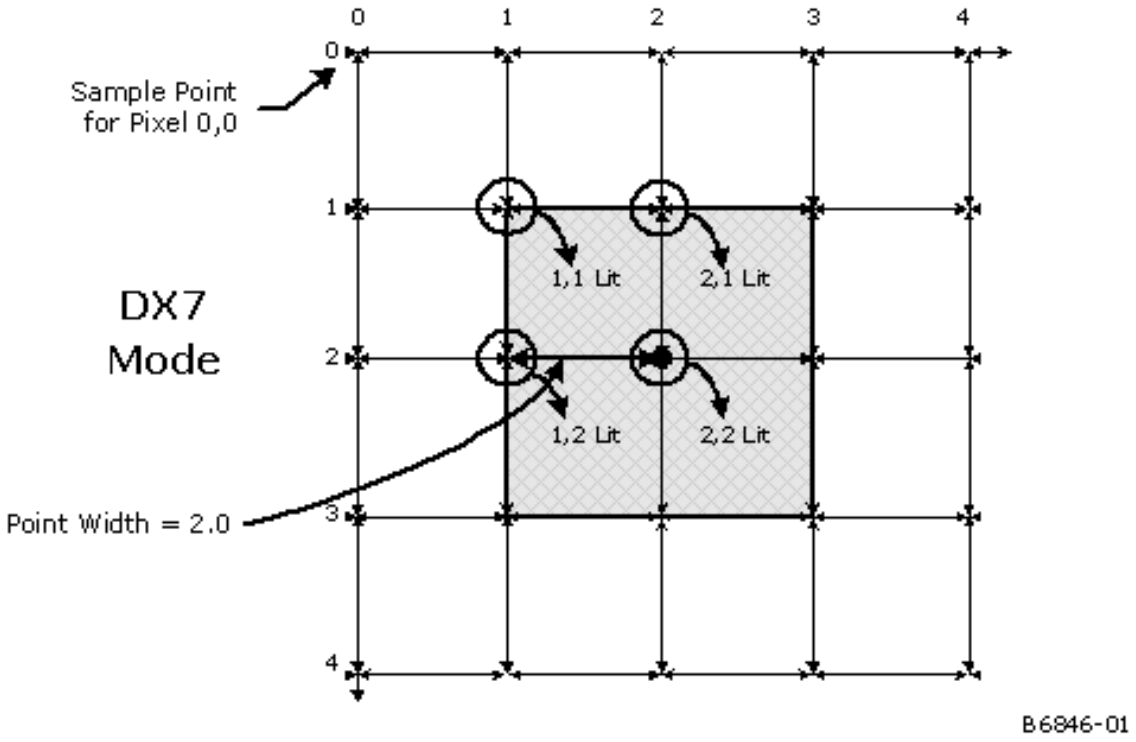
Destination Origin Bias



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

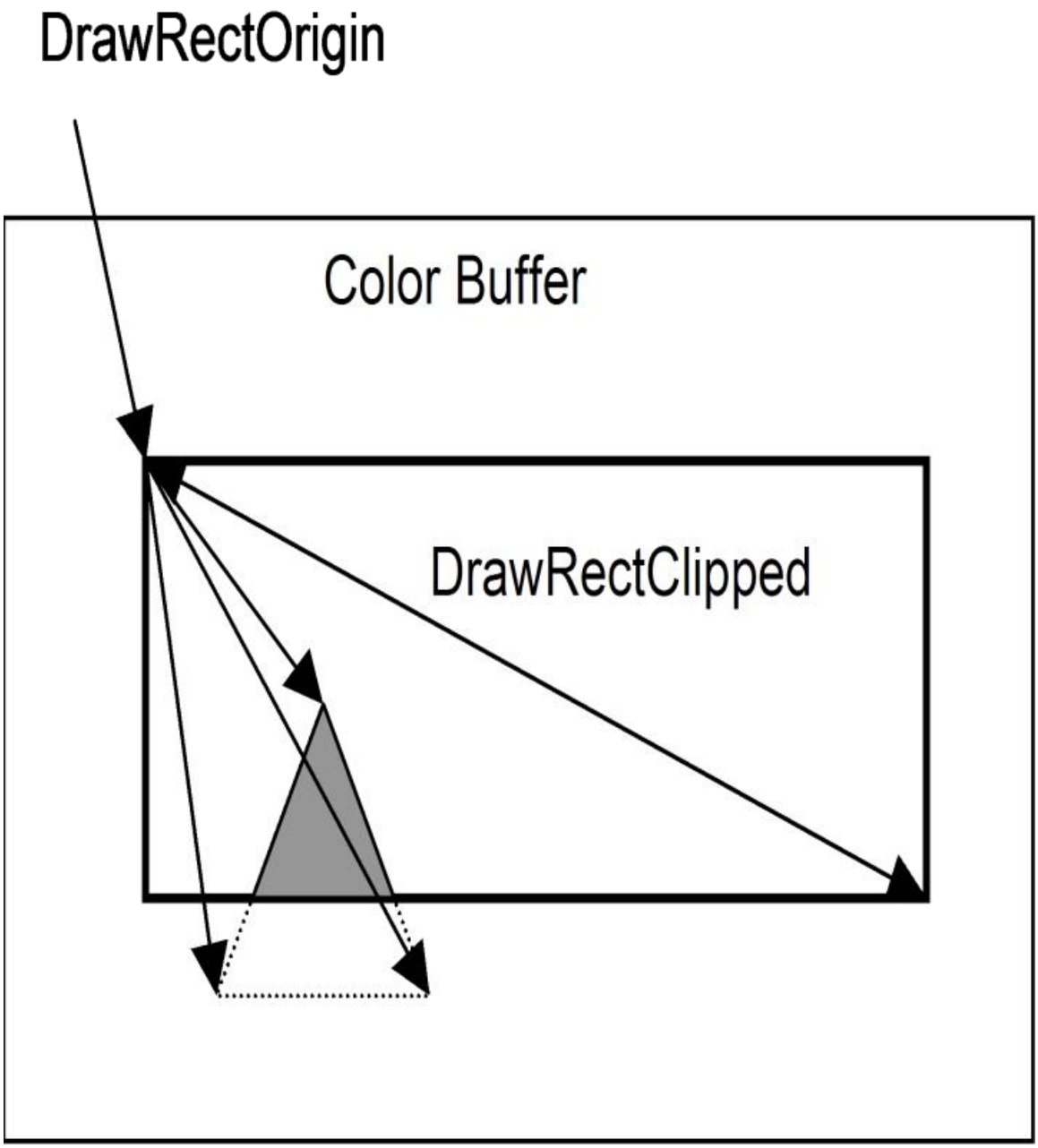


Drawing Rectangle Offset Application

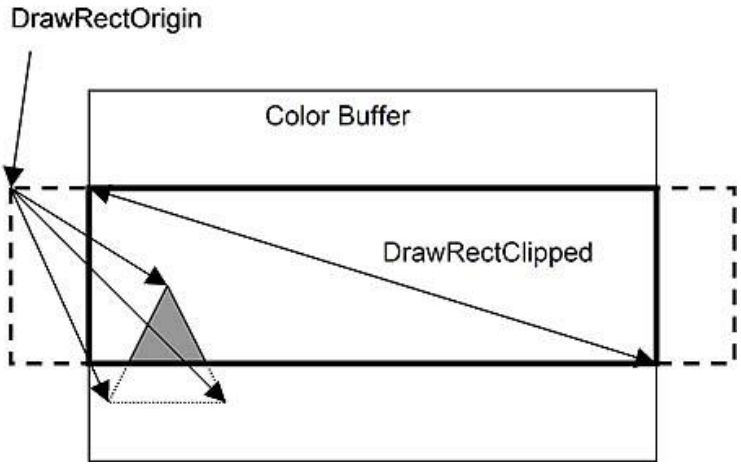
The Drawing Rectangle Offset subfunction offsets the object's vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window's or viewport's upper-left corner.

Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the "clipped" draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.

Onscreen Draw Rectangle



Partially-offscreen Draw Rectangle



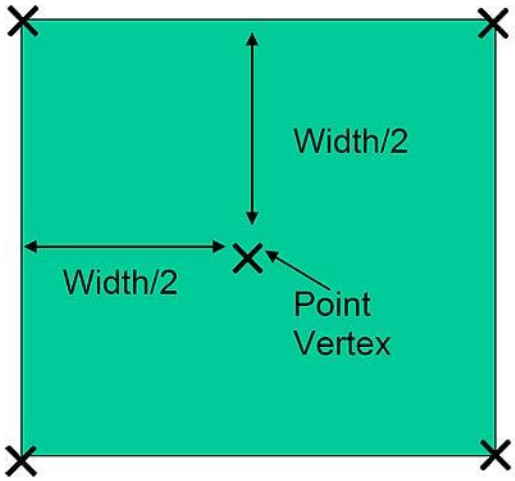
3DSTATE_DRAWING_RECTANGLE

Point Width Application

This stage of the pipeline applies only to 3DOBJ_POINT objects. Here the point object is converted from a single vertex to four vertices located at the corners of a square centered at the point's X,Y position. The width and height of the square are specified by a *point width* parameter. The **Point Width Source** value in SF_STATE determines the source of the point width parameter: the point width is either taken from the **Point Width** value programmed in SF_STATE or the PointWidth specified with the vertex (as read back from the vertex VUE earlier in the pipeline).

The corner vertices are computed by adding and subtracting one half of the point width. *Point Width Application*.

Point Width Application

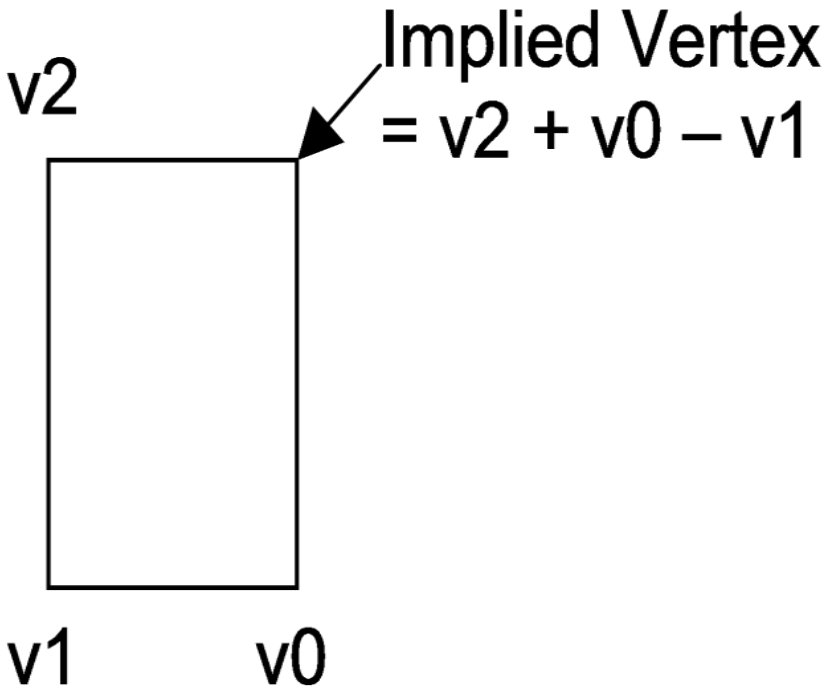


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

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 Object Culling 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

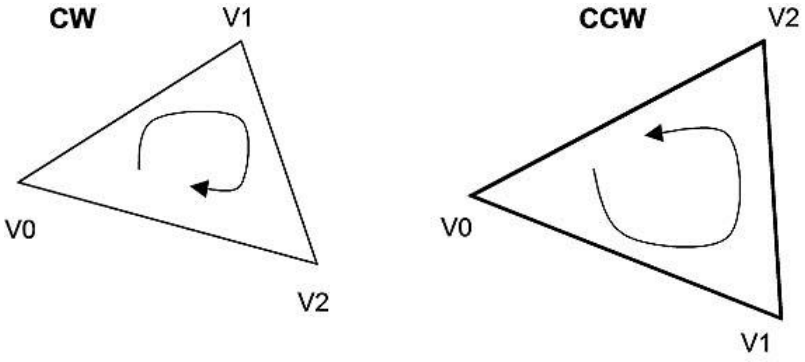
At this stage of the pipeline, 3DOBJ_TRIANGLE objects can be optionally discarded based on the "face orientation" of the object. This culling operation does not apply to the other object types.

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 the triangle vertex's 2D (X,Y) screen space position when traversed from v0 to v1 to v2. That traversal proceeds in either a clockwise (CW) or counter-clockwise (CCW) direction. The "winding order" of a triangle is defined by the the triangle vertex's 2D

(X,Y) screen space position when traversed from v0 to v1 to v2. That traversal will proceed in either a clockwise (CW) or counter-clockwise (CCW) direction. A degenerate triangle is considered "backfacing", regardless of the FrontWinding state.

Triangle Winding Order



The **Front Winding** state variable in SF_STATE controls whether CW or CCW triangles are considered as having a "front-facing" orientation (at which point non-front-facing triangles are considered "back-facing"). The internal variable *invertOrientation* associated with the triangle object is then used to determine whether the orientation of a that triangle should be inverted. Recall that this variable is set in the Primitive Decomposition stage to account for the alternating orientations of triangles in strip primitives resulting form the ordering of the vertices used to process them.

The **Cull Mode** state variable in SF_STATE specifies how triangles are discarded according to their resultant orientation. See Degenerate Objects.

Cull Mode

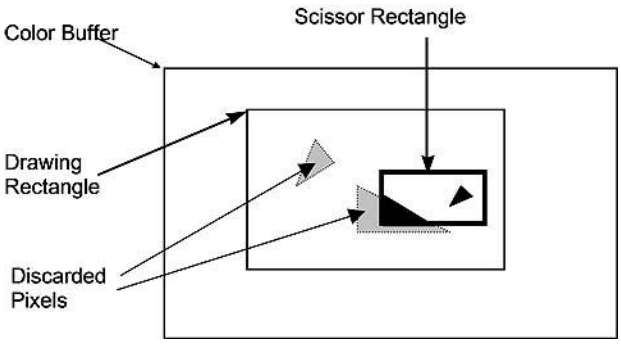
CullMode	Definition
CULLMODE_NONE	The face culling operation is disabled.
CULLMODE_FRONT	Triangles with "front facing" orientation are discarded.
CULLMODE_BACK	Triangles with "back facing" orientation are discarded.
CULLMODE_BOTH	All triangles are discarded.

Scissor Rectangle Clipping

A *scissor* operation can be used to restrict the extent of rendered pixels to a screen-space aligned rectangle. If the scissor operation is enabled, portions of objects falling outside of the intersection of the scissor rectangle and the clipped draw rectangle are clipped (pixels discarded).

The scissor operation is enabled by the **Scissor Rectangle Enable** state variable in SF_STATE. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding SF_VIEWPORT structure. Up to 16 structures are supported. The **Scissor Rectangle X,Y Min,Max** fields of the SF_VIEWPORT structure defines a scissor rectangle as a rectangle in integer pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. The scissor rectangle is defined relative to the Drawing Rectangle to better support the OpenGL API. (OpenGL specifies the "Scissor Box" in window-

relative coordinates). This allows instruction buffers with embedded Scissor Rectangle definitions to remain valid even after the destination window (drawing rectangle) moves.



Specifying either scissor rectangle xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null scissor rectangle).

Viewport Extents Test

Viewport extents test can be used to restrict the extent of rendered pixels to the viewport extents. If this operation is enabled, portion of the objects falling outside of the intersection of the scissor rectangle (if enabled) and the clipped draw rectangle and viewport extents are clipped (pixels discarded). This operation similar to the scissor test except both have different enables and the viewport extents can be programmed to the fractional float values.

This operation is enabled by the **View Transform Enable** state variable in **SF_STATE**. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding **SF_CLIP_VIEWPORT** structure. Up to 16 structures are supported. The X/Y **Min/Max ViewPort** fields of the **SF_CLIP_VIEWPORT** structure defines viewport extents as a rectangle in float screen pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. Please note that these co-ordinates can be fractional values and hardware will do appropriate rounding and convert it to integer pixel co-ordinates (floor rouding used). This **View Transform Enable** state also controls the viewport transform so appropriate the viewport transform coefficients need to be populated in the SF_CLIP_VIEPWORT structure along with the viewport extents.

Final clip rectangle used to define the rendering area will now depend on three rectangles namely drawing rectangle, Scissor rectangle, Viewport Extents. If both **Scissor Rectangle Enable** and **View transform enable** are set then intersection of all rectangles (Viewport extents, Scissor rectangle, Draw rectangle) becomes final clip rectangle, while If only **Scissor Rectangle Enable** is enabled then the intersection of (Scissor rectangle, Draw rectangle) becomes final clip rectangle. If only **View transform enable** is enabled then intersection of (Viewport extents, Draw rectangle) become the final clip rectangle, while If none is enabled then (Draw rectangle) is the final clip rectangle.

Specifying either viewport extents xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null viewport).

Line Rasterization

The device supports three styles of line rendering: *zero-width* (*cosmetic*) lines, *non-antialiased* lines, and *antialiased* lines. Non-antialiased lines are rendered as a polygon having a specified width as measured parallel to the major axis of the line. Antialiased lines are rendered as a rectangle having a specified width measured perpendicular to the line connecting the vertices.

The functions required to render lines are split between the SF and WM units. The SF unit is responsible for computing the overall geometry of the object to be rendered, including the pixel-exact bounding box, edge equations, etc., and therefore is provided with the screen-geometry-related state variables. The WM unit performs the actual scan conversion, determining the exact pixels included/excluded and coverage values for anti-aliased lines.

Zero-Width (Cosmetic) Line Rasterization

Note: The specification of zero-width line rasterization would be more correctly included in the WM Unit chapter, though is being included here to keep it with the rasterization details of the other line types.

When the **Line Width** is set to zero, the device will use special rules to rasterize zero-width ("cosmetic") lines. The **Anti-Aliasing Enable** state variable is ignored when **Line Width** is zero.

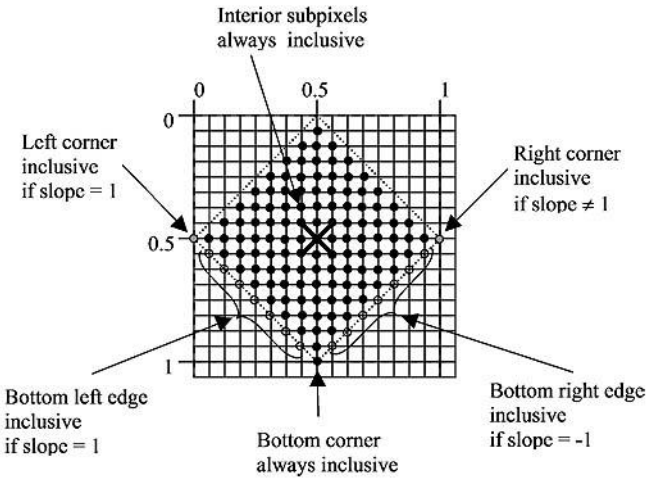
When the **LineWidth** is set to zero, the device will use special rules to rasterize "cosmetic" lines. The rasterization rules also comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

The GIQ rules basically intersect the directed, ideal line connecting two endpoints with an array of diamond-shaped areas surrounding pixel sample points. Wherever the line exits a diamond (including passing through a diamond), the corresponding pixel is lit. Special rules are used to define the subpixel locations that are considered interior to the diamonds, as a function of the slope of the line. When a line ends in a diamond (and therefore does not exit that diamond), the corresponding pixel is not drawn. When a line starts in a diamond and exits that diamond, the corresponding pixel is drawn.

GIQ (Diamond) Sampling Rules – Legacy Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is ENABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered "interior" to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range (1, +infinity] or [-infinity, -1) touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) – where if the right corner was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside" for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.
4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

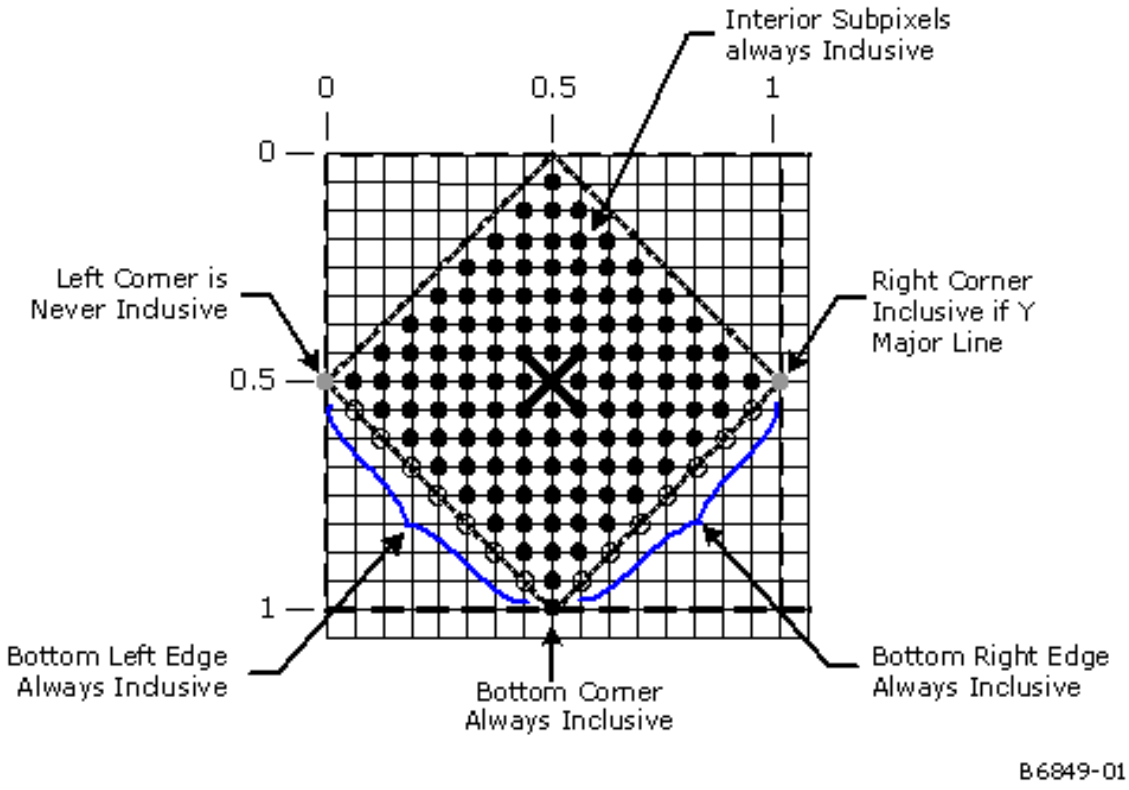
The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```
      delta_x                          = point.x - sample.x
      delta_y                          = point.y - sample.y
      distance                   = abs(delta_x) + abs(delta_y)
      interior                   = (distance < 0.5)
      bottom_corner              = (delta_x == 0.0) && (delta_y == 0.5)
      left_corner                = (delta_x == -0.5) && (delta_y == 0.0)
      right_corner               = (delta_x == 0.5) && (delta_y == 0.0)
      bottom_left_edge           = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
      bottom_right_edge          = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

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

GIQ (Diamond) Sampling Rules – DX10 Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is DISABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered "interior" to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as -1 <= slope <= 1).** Including the right corner subpixel ensures that lines with slopes in the range (>1, +infinity] or [-infinity, <-1) touch a diamond even when they cross exactly between pixel diamonds.
3. **The left corner subpixel is never inclusive.** For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
4. **The subpixels along the bottom left edge are always inclusive.** This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would cause the pixel to be lit twice – breaking the rule that connected line segments

should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside", the pixel is only lit when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

5. **The subpixels along the bottom right edge are always inclusive.** Same as case as (4), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (XMajor).

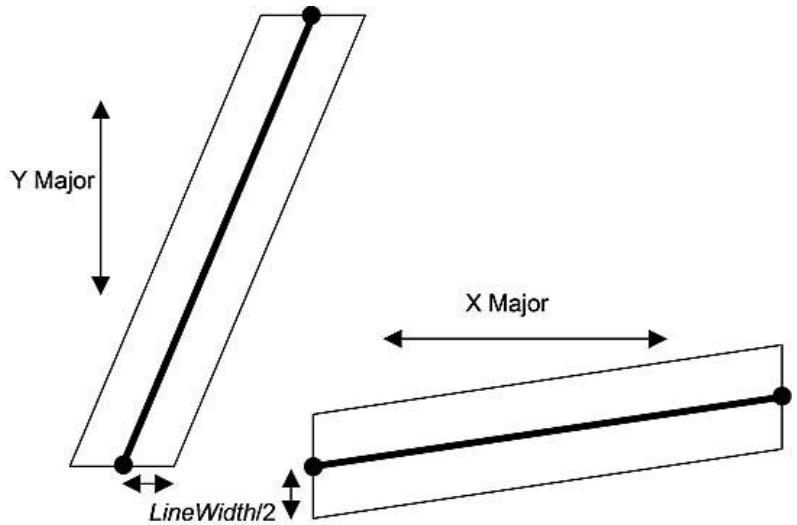
```
delta_x                         = point.x - sample.x
delta_y                         = point.y - sample.y
distance                        = abs(delta_x) + abs(delta_y)
interior                        = (distance < 0.5)
bottom_corner                         = (delta_x == 0.0)  && (delta_y == 0.5)
left_corner                     = (delta_x == -0.5) && (delta_y == 0.0)
right_corner                    = (delta_x == 0.5)  && (delta_y == 0.0)
bottom_left_edge                = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge               = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

inside = interior || bottom_corner || (!XMajor && right_corner) || ( bottom_left_edge)
|| ( bottom_right_edge)
```

Non-Antialiased Wide Line Rasterization

Non-anti-aliased, non-zero-width lines are rendered as parallelograms that are centered on, and aligned to, the line joining the endpoint vertices. Pixels sampled interior to the parallelogram are rendered; pixels sampled exactly on the parallelogram edges are rendered according to the polygon "top left" rules.

The parallelogram is formed by first determining the major axis of the line (diagonal lines are considered x-major). The corners of the parallelogram are computed by translating the line endpoints by +/-(**Line Width** / 2) in the direction of the minor axis, as shown in the following diagram.

Non-Antialiased Line Rasterization

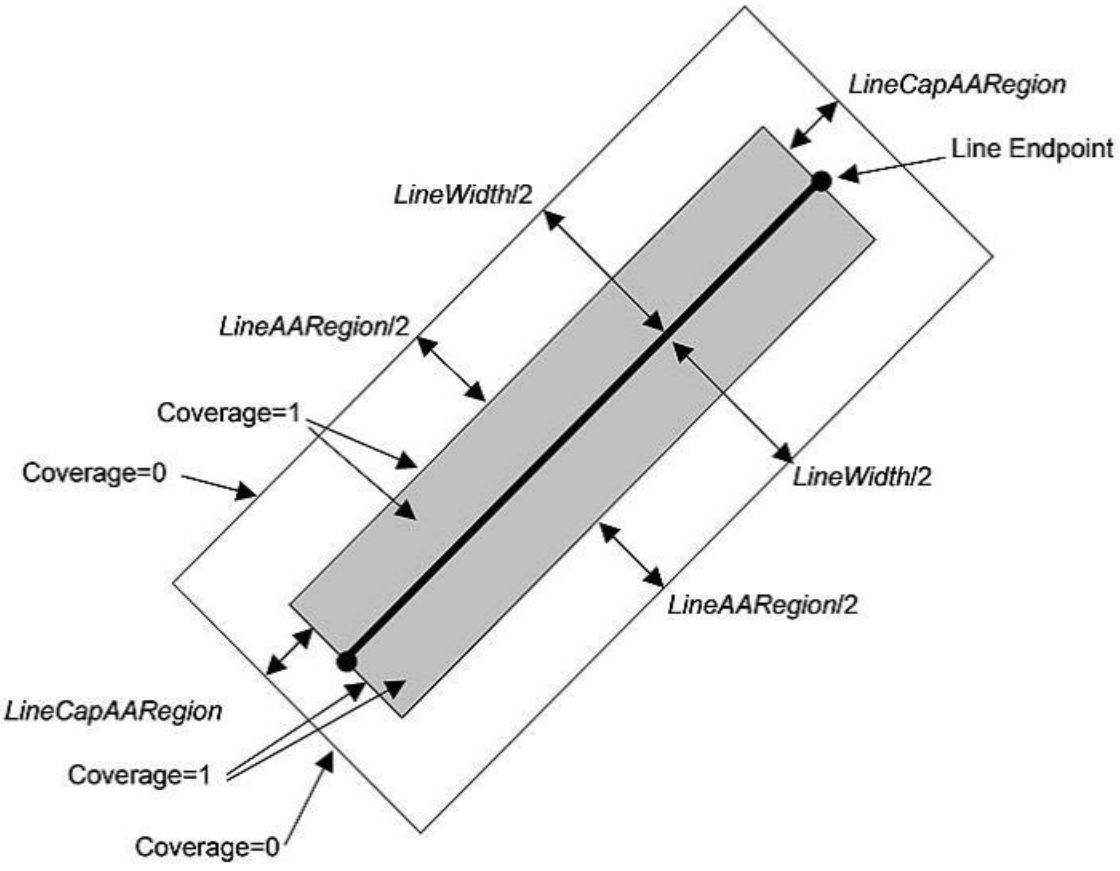


Anti-Aliased Line Rasterization

Anti-aliased lines are rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices. For each pixel in the rectangle, a fractional coverage value (referred to as Antialias Alpha) is computed – this coverage value is normally used to attenuate the pixel's alpha in the pixel shader thread. The resultant alpha value is therefore available for use in those downstream pixel pipeline stages to generate the desired effect (e.g., use the attenuated alpha value to modulate the pixel's color, and add the result to the destination color, etc.). Note that software is required to explicitly program the pixel shader and pixel pipeline to obtain the desired anti-aliasing effect – the device simply makes the coverage-attenuated pixel alpha values available for use in the pixel shader.

The dimensions of the rendered rectangle, and the parameters controlling the coverage value computation, are programmed via the **Line Width**, **Line AA Region**, and **Line Cap AA Region** state variables, as shown below. The edges parallel to the line are located at the distance (*LineWidth*/2) from the line (measured in screen pixel units perpendicular to the line). The end-cap edges are perpendicular to the line and located at the distance (*LineCapAARegion*) from the endpoints.

Anti-aliased Line Rasterization



Along the parallel edges, the coverage values ramp from the value 0 at the very edges of the rectangle to the value 1 at the perpendicular distance (*LineAARegion*/2) from a given edge (in the direction of the line). A pixel's coverage value is computed with respect to the closest edge. In the cases where

(*LineAARegion*/2) < (*LineWidth*/2), this results in a region of fractional coverage values near the edges of the rectangle, and a region of "fully-covered" coverage values (i.e., the value 1) at the interior of the line. When (*LineAARegion*/2) == (*LineWidth*/2), only pixel sample points falling exactly on the line can generate fully-covered coverage values. If (*LineAARegion*/2) > (*LineWidth*/2), no pixels can be fully-covered (it is expected that this case is not typically desired).

Along the end cap edges, the coverage values ramp from the value 1 at the line endpoint to the value 0 at the cap edge – itself at a perpendicular distance (*LineCapAARegion*) from the endpoint. Note that, unlike the line-parallel edges, there is only a single parameter (*LineCapAARegion*) controlling the extension of the line at the end caps and the associated coverage ramp.

The regions near the corners of the rectangle have coverage values influenced by distances from both the line-parallel and end cap edges – here the two coverage values are multiplied together to provide a composite coverage value.

The computed coverage value for each pixel is passed through the Windower Thread Dispatch payload. The Pixel Shader kernel should be passed (unmodified) by the shader to the Render Cache as part of it's output message.

SF Pipeline State Summary

3DSTATE_RASTER

3DSTATE_RASTER

Signal	SF_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table: WM_INT::Multisample Rasterization Mode in 3D Pipeline Windower > Windower Pipelined State > 3DSTATE_WM > 3DSTATE_WM

Signal	SF_INT::Global Depth Offset Enable Solid
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Solid? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT:: Global Depth Offset Enable Wireframe
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Wireframe? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT:: Global Depth Offset Enable Point
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Point? ((3DSTATE_RASTER::Global De IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disab

3DSTATE_SF

The state used by the SF stage is defined by this inline state packet.

3DSTATE_SF

The SF Unit also receives 3DSTATE_RASTER. It also receives 3DSTATE_INT which is transparent to SW. 3DSTATE_INT provides 3DSTATE_WM, 3DSTATE_WM_HZ_OP, 3DSTATE_DETPH_BUFFER, and 3DSTATE_MULTISAMPLE fields.

Signal	SF_INT::Number of Multisamples
Description	Set the number of multisamples.
Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP::Number of Multisamples : 3DSTATE_MULTISAMPLE::Number of Multisamples

Signal	SF_INT::Pixel Position Offset Enable
Description	Enables the device to offset pixel positions by 0.5 both in horizontal and vertical directions.

Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP:: Pixel Position Offset Enable: 3DSTATE_MULTISAMPLE:: Pixel Position Offset Enable

Signal	SF_INT::Pixel Position Offset
Description	Causes the device to offset pixel positions by 0.5 both in horizontal and vertical directions. It is to be noted this is done to adjust the pixel co-ordinate system to DX9 like, so any screen space rectangles (eg: HiZ Clear, Resolve etc) generated internally by driver in this mode needs to be aware of this offset adjustment and send the rectangles according to alignment restriction taking this offset adjustment into consideration.
Formula	= (SF_INT::Number of Multisamples >1) && (3DSTATE_MULTISAMPLE:: Pixel Location == PIXLOC_UL_CORNER) && SF_INT::Pixel Position Offset Enable

Signal	SF_INT: Global Depth Offset Enable Solid
Description	Set the number of multisamples
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Solid ? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) ‖ (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT: Global Depth Offset Enable Wireframe
Description	Set the number of multisamples
Fromula	= (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) ? 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 for , a number of state fields in 3DSTATE_SF are utilized to control interpolation setup.

Number of SF Output Attributes sets the number of attributes that will be output from the SF stage, not including position. This can be used to specify up to 32, and may differ from the number of input attributes. The number of input attributes is derived from the **Vertex URB Entry Read Length** field. Note that this field is also used to specify whether swizzling is to be performed on Attributes 0-15 or Attributes 16-32. See the state field definition for details.

Attribute Swizzling

The first or last set of 16 attributes can be swizzled according to certain state fields. **Attribute Swizzle Enable** enables the swizzling for all 16 of these attributes, and each of the attributes has a 2-bit **Swizzle Select** field that controls swizzling with the following settings:

- INPUTATTR – This attribute is sourced from AttrInputReg[SourceAttribute].
- INPUTATTR_FACING – This attribute is sourced from AttrInputReg[SourceAttribute] if the object is front-facing, otherwise it is sourced from AttrInputReg[SourceAttribute+1].
- INPUTATTR_W – This attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination).
- INPUTATTR_FACING – If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination). If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. WYZW.

Each of the first or last set of 16 attributes also has a 5-bit **Source Attribute** field which specify, per output attribute (not component), which input attribute sources the output attribute when INPUTATTR is selected for **Swizzle Select**. A **Source Attribute** value of 0 corresponds to the 128-bit attribute immediately following the vertex 4D position. If INPUTATTR_FACING is selected, this specifies the first of two consecutive (front,back) input attributes, where the SourceAttribute value can be an odd or even number (just not 31, as that would place the back-face input attribute past the end of the input max complement of input attributes).

Constant overriding is also available for the first or last set of 16 attributes. Each attribute has a **Constant Source** field which specifies the constant values per swizzled attribute, with the following settings available:

- XYZW = 0000
- XYZW = 0001
- XYZW = 1111

Each channel of each attribute has a **Component Override** field to control whether the corresponding channel is overridden with the constant value defined in **Constant Source**.

Interpolation Modes

All 32 attributes have a **Constant Interpolation Enable** state field bit to specify whether all components of the post-swizzled attribute are to be interpolated as constant values (not varying over the pixels of the object). If set, the attribute at the provoking vertex is copied to a0, and a1 and a2 are set to zero – this results in a constant interpolation of the provoking vertex value. If clear, the attribute is linearly interpolated. Attributes 0-15 are further subjected to Wrap Shortest processing on a per-component basis, via the **Attribute WrapShortest Enables** state bitfields. WrapShortest processing modifies the a1 and/or a2 values depending on attribute deltas. All

The table below indicates the output values of a0, a1, and a2 depending on interpolation mode settings.

	a0	a1	a2
Constant	A0	0.0	0.0
Linear	A0	A1-A0	A2-A0
Wrap Shortest	A0	(A1-A0)+1 (A1-A0) <= -0.5 (A1-A0)-1 (A1-A0) >= 0.5 (A1-A0) otherwise	(A2-A0)+1 (A2-A0) <= -0.5 (A2-A0)-1 (A2-A0) >= 0.5 (A2-A0) otherwise

Point Sprites

Normally all vertex attributes (including texture coordinates) other than position are simply replicated from the incoming point center vertex to the generated point object (corner) vertices. However, both DX9 and OGL support “sprite points”, where some/all texture coordinates are replaced with full-scale 2D texture coordinates.

A 32-bit **PointSprite TextureCoordinate Enable** bit mask controls whether the corresponding vertex attribute is to be replaced by a sprite point texture coordinate. The global (not per-attribute) **Point Sprite TextureCoordinate Origin** field controls how the point object vertex (top/bottom, left/right) texture coordinates are generated:

UPPERLEFT	Left	Right
Top	(0,0,0,1)	(1,0,0,1)
Bottom	(0,1,0,1)	(1,1,0,1)

LOWERLEFT	Left	Right
Top	(0,1,0,1)	(1,1,0,1)
Bottom	(0,0,0,1)	(1,0,0,1)

The state used by “setup backend” is defined by the following inline state packet.

3DSTATE_SBE

The state used by “setup backend” is defined by the following inline state packet.

3DSTATE_SBE_SWIZ

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

Signal	SBE_INT::Vertex URB Entry Read Length
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::ThreadDispatchEnable
Description	This bit, if set, indicates that it is possible for a PS thread to modify a render target.
Formula	= (3DSTATE_WM::ForceThreadDispatch == ON) || ((3DSTATE_WM::ForceThreadDispatch != OFF) && ! WM_INT::WM_HZ_OP && 3DSTATE_PS_EXTRA::PixelShaderValid && ((!3DSTATE_PS_EXTRA::PixelShaderDoesNotWriteRT && 3DSTATE_PS_BLEND::HasWriteableRT) || (3DSTATE_PS_EXTRA::PixelShaderHasUAV) || WM_INT:: Pixel Shader Kill Pixel || (WM_INT::Pixel Shader Computed Depth Mode != PSCDEPTH_OFF &&

	(WM_INT::Depth Test Enable || WM_INT::Depth Write Enable)) || (3DSTATE_PS_EXTRA::Computed Stencil && WM_INT::Stencil Test Enable) || (3DSTATE_WM::EDSC_Mode == 1 && (WM_INT::Depth Test Enable || WM_INT::Depth Write Enable || WM_INT::Stencil Test Enable)) || (WM_INT::RT Independent Rasterization Enable)))

Signal	WM_INT::Pixel Shader Computed Depth Mode
Description	This field specifies the computed depth mode for the pixel shader.
Formula	= (3DSTATE_PS_EXTRA::ForceComputedDepth == Force) ? 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode : (WM_INT::WM_HZ_OP || WM_INT::RT Independent Rasterization Enable) ? PSCDEPTH_OFF: 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode

Signal	WM_INT:: Pixel Shader Computed Stencil
Description	This field specifies the computed stencil mode for the pixel shader.
Formula	= (WM_INT::WM_HZ_OP) ? 0 : 3DSTATE_PS_EXTRA::Computed Stencil

Signal	WM_INT::Pixel Shader Uses Source Depth
Description	This bit, if ENABLED, indicates that the PS kernel requires the source depth value (vPos.z) to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source Depth

Signal	WM_INT::Pixel Shader Uses Source W

Description	This bit, if ENABLED, indicates that the PS kernel requires the interpolated source W value (vPos.w) to be passed in the payload
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source W

Signal	WM_INT::Pixel Shader Uses Input Coverage Mask
Description	This bit, if ENABLED, indicates that the PS kernel requires the input coverage mask to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Input Coverage Mask

Signal	WM_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table below: WM_INT::Multisample Rasterization Mode

WM_INT::Multisample Rasterization Mode

3DSTATE_RASTER :: Force Multisampling	Force	Force	Force	Force	Normal	Normal	Normal
3DSTATE_RASTER :: DX Multisample Rasterization Mode	MSRASTMODE_OFF_PIXEL	MSRASTMODE_OFF_PATTERN	MSRASTMODE_ON_PIXEL	MSRASTMODE_ON_PATTERN	*	*	*
WM_INT::WM_HZ_OP	*	*	*	*	True	True	False
3DSTATE_WM_HZ_OP:: Number of Multisamples	*	*	*	*	> NUMSAMPLES_1	NUMSAMPLES_1	*
WM_INT::Multisample Rasterization Mode	OFF_PIXEL	OFF_PATTERN	ON_PIXEL	ON_PATTERN	ON_PATTERN	ON_PIXEL	Determined from Table 1 in 3D Pipeline Windower

							> Multisampling > Multisample Modes/State)

Note: OFF_PIXEL, OFF_PATTERN, ON_PIXEL, ON_PATTERN modes are described in 3D Pipeline Windower > Multisampling > Multisample Modes/State.

Signal	WM_INT::Multisample Dispatch Mode
Description	This bit, determines how PS threads are dispatched
Formula	= (WM_INT::RT Independent Rasterization Enable)? PerPixel: (3DSTATE_PS_EXTRA::PixelShaderIsPerSample) ? PerSample : PerPixel

Signal	WM_INT::Pixel Shader Kill Pixel
Description	This bit, if ENABLED, indicates that the PS kernel or color calculator has the ability to kill (discard) pixels or samples, *other than due to depth or stencil testing*.
Formula	= (3DSTATE_WM::ForceKillPixel == ON) || ((3DSTATE_WM::ForceKillPixel != Off) && ! WM_INT::WM_HZ_OP && ! 3DSTATE_WM::EDSC_Mode == 2 && (WM_INT::Depth Write Enable || WM_INT::Stencil Write Enable) && (3DSTATE_PS_EXTRA::PixelShaderKillsPixels || 3DSTATE_PS_EXTRA:: **oMask Present to RenderTarget** || 3DSTATE_PS_BLEND::AlphaToCoverageEnable || 3DSTATE_PS_BLEND::AlphaTestEnable || 3DSTATE_WM_CHROMAKEY::ChromaKeyKillEnable

	))

Signal	WM_INT::Early Depth/Stencil Control
Description	This field specifies the behavior of early depth/stencil test.
Formula	= (WM_INT::WM_HZ_OP) ? EDSC_NORMAL : WM_INT::RT Independent Rasterization Enable ? EDSC_PSEXEC : 3DSTATE_WM::Early Depth/Stencil Control

Signal	WM_INT::RT Independent Rasterization Enable
Description	Enables Render Target Independent Rasterization.
Formula	= (WM_INT::WM_HZ_OP ? Disable : (3DSTATE_RASTER::ForceSampleCount != NUMRASTSAMPLES_0) ? Enable : Disable

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

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

Signal	WM_INT::WM_HZ_OP

Description	Enables WM_HZ_OP
Formula	= (3DSTATE_WM_HZ_OP::DepthBufferClear || 3DSTATE_WM_HZ_OP::DepthBufferResolve || 3DSTATE_WM_HZ_OP::Hierarchical Depth Buffer Resolve Enable || 3DSTATE_WM_HZ_OP::StencilBufferClear) ? Enable : Disable

Signal	WM_INT:: Pixel Location
Description	Sets the input pixel location to Center if UL and doing multisampling
Formula	(3DSTATE_MULTISAMPLE::Pixel Location && 3DSTATE_MULTISAMPLE::Pixel Position Offset Enable && WM_MULTISAMPLE_INT::Number of Multisamples > 0) ? 0 : 3DSTATE_MULTISAMPLE::Pixel Location

3DSTATE_SAMPLE_MASK

The following inline state packets define the sample mask state used by the windower stage for different generations.

State Packets
3DSTATE_SAMPLE_MASK

Signal	WM_INT:: Sample Mask Enable
Description	Sets Sample Mask used in rasterization

Formula	Switch(WM_MULTISAMPLE_INT::Number of Multisamples
	{
	Case NUMSAMPLES_1: WM_INT:: Sample Mask Enable = 0x0001; break;
	Case NUMSAMPLES_2: WM_INT:: Sample Mask Enable = 0x0003; break;
	Case NUMSAMPLES_4: WM_INT:: Sample Mask Enable = 0x000F; break;
	Case NUMSAMPLES_8: WM_INT:: Sample Mask Enable = 0x00FF; break;
	Add this additional case:
	Case NUMSAMPLES_16: WM_INT:: Sample Mask Enable = 0xFFFF; break;
	}

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

3DSTATE_WM_CHROMAKEY

3DSTATE_WM_HZ_OP

State Restrictions

State	Restriction
3DSTATE_PS::Render Target Fast Clear Enable	Must be disabled
3DSTATE_PS:: Render Target Resolve Enable	Must be disabled
3DSTATE_WM:: Legacy Depth Buffer Clear	Must be disabled
3DSTATE_WM:: Legacy Depth Buffer Resolve	Must be disabled
3DSTATE_WM:: Legacy Hierarchical Depth Buffer Resolve Enable	Must be disabled
3DSTATE_MULTISAMPLE::Pixel Location	Must be set according to the API being used.
3DSTATE_CLEAR_PARAMS ::Depth Clear Value	Must be programmed according to the API when 3DSTATE_WM_HZ_OP::Depth Buffer Clear is set
3DSTATE_CLEAR_PARAMS ::Depth Clear Value Valid	Must be enabled when 3DSTATE_WM_HZ_OP::Depth Buffer Clear is set

State Overrides

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable	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) ?	(3DSTATE_WM_HZ_OP ::NumberOfSamples > 1) ?	(3DSTATE_WM_HZ_OP ::NumberOfSamples > 1) ?	(3DSTATE_WM_HZ_OP ::NumberOfSamples > 1) ?

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable	Hierarchical Depth Buffer Resolve Enable
	ON_PATTERN : ON_PIXEL	ON_PATTERN : ON_PIXEL	ON_PATTERN : ON_PIXEL	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::StencilTestEnable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::StencilWriteEnable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::DepthTestEnable	Disable	Disable	Enable	Disable
WM_INT::DepthWriteEnable	Depth buffer Clear ? Enable : Disable	Enable	Enable	Enable
WM_INT::DepthTestFunction	NEVER	NEVER	NEVER	NEVER

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable	Hierarchical Depth Buffer Resolve Enable
WM_INT::StencilTestFunction	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:: Sample Mask	3DSTATE_WM_HZ_OP:: Sample Mask	3DSTATE_WM_HZ_OP:: Sample Mask	3DSTATE_WM_HZ_OP:: Sample Mask
WM_INT::Early Depth/Stencil Control	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL
WM_INT:: Full	Depth buffer clear ?	WM_HZ_OP:: Full	Disable	Disable

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable	Hierarchical Depth Buffer Resolve Enable
Surface Depth Clear	WM_HZ_OP:: Full Surface Depth Clear : Disable	Surface Depth Clear		
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::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	= 3DSTATE_STENCIL_BUFFER::STENCIL_BUFFER_ENABLE && 3DSTATE_DEPTH_BUFFER::STENCIL_WRITE_ENABLE && ((WM_INT::WM_HZ_OP ? Use the WM_INT::StencilWriteEnable from WM_HZ_OP table : WM_INT::StencilTestEnable && 3DSTATE_WM_DEPTH_STENCIL::StencilBufferWriteEnable))

Signal	WM_INT::StencilTestEnable
Description	Enables Stencil Test
Formula	= 3DSTATE_STENCIL_BUFFER::STENCIL_BUFFER_ENABLE && (WM_INT::WM_HZ_OP ? Use the WM_INT::StencilTestEnable from WM_HZ_OP table : (3DSTATE_WM_DEPTH_STENCIL::StencilTestEnable && !WM_INT::RT Independent Rasterization Enable))

Signal	WM_INT::DepthTestEnable
Description	Enables Depth Test

Formula	= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL) && (WM_INT::WM_HZ_OP ? Use the WM_INT::DepthTestEnable from WM_HZ_OP table : (3DSTATE_WM_DEPTH_STENCIL::DepthTestEnable && ! WM_INT::RT Independent Rasterization Enable))

Signal	WM_INT::DepthWriteEnable
Description	Enables Depth Write
Formula	= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL) && 3DSTATE_DEPTH_BUFFER::DEPTH_WRITE_ENABLE && (WM_INT::WM_HZ_OP ? Use the WM_INT::DepthWriteEnable from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::DepthWriteEnable)

Signal	WM_INT::DepthTestFunction
Description	Depth Test Function
Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::DepthTestFunction from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::DepthTestFunction

Signal	WM_INT::StencilTestFunction
Description	Stencil Test Function
Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::StencilTestFunction from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::StencilTestFunction

Signal	WM_INT::StencilPassDepthPassOp
Description	StencilPassDepthPassOp

Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::StencilPassDepthPassOp from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::StencilPassDepthPassOp

Signal	WM_INT::Stencil Test Mask
Description	Stencil test Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Stencil Test Mask

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

Signal	WM_INT::BackFace Stencil Test Mask
Description	Stencil test Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL:: Backface Stencil Test Mask

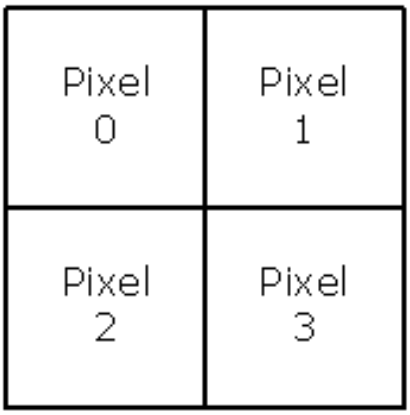
Signal	WM_INT:: BackFace Stencil Write Mask
Description	Stencil Write Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Backface Stencil Write Mask

Rasterization

The WM unit uses the setup computations performed by the SF unit to rasterize objects into the corresponding set of pixels Most of the controls regarding the screen-space geometry of rendered objects are programmed via the SF unit.

The rasterization process generates pixels in 2x2 groups of pixels called *subspans* (see *Pixels with a SubSpan* below) which, after being subjected to various inclusion/discard tests, are grouped and passed to spawned Pixel Shader (PS) threads for subsequent processing Once these PS threads are spawned, the WM unit provides only bookkeeping functions on the pixels Note that the WM unit can proceed on to rasterize subsequent objects while PS threads from previous objects are still executing.

Pixels with a SubSpan



B6850-01

Drawing Rectangle Clipping

The Drawing Rectangle defines the maximum extent of pixels which can be rendered Portions of objects falling outside the Drawing Rectangle will be clipped (pixels discarded) Implementations will typically discard objects falling completely outside of the Drawing Rectangle as early in the pipeline as possible There is no control to turn off Drawing Rectangle clipping – it is unconditional.

For the purposes of clipping, the Drawing Rectangle must itself be clipped to the destination buffer extents (The Drawing Rectangle Origin, used to offset relative X,Y coordinates earlier in the pipeline, is permitted to lie offscreen). The **Clipped Drawing Rectangle X,Y Min,Max** state variables (programmed via 3DSTATE_DRAWING_RECTANGLE – See *SF Unit*) defines the intersection of the Drawing Rectangle and the Color Buffer It is specified with non-negative integer pixel coordinates relative to the Destination Buffer upper-left origin.

Pixels with coordinates outside of the Drawing Rectangle cannot be rendered (i.e., the rectangle is inclusive) For example, to render to a full-screen 1280x1024 buffer, the following values would be required: Xmin=0, Ymin=0, Xmax=1279 and Ymax=1023

For “full screen” rendering, the Drawing Rectangle coincides with the screen-sized buffer For “front-buffer windowed” rendering it coincides with the destination “window”.

Line Rasterization

See *SF Unit* chapter for details on the screen-space geometry of the various line types.

Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

3DSTATE_AA_LINE_PARAMS

3DSTATE_AA_LINE_PARAMETERS

The slope and bias values should be computed to closely match the reference rasterizer results Based on empirical data, the following recommendations are offered:

The final alpha for the center of the line needs to be 148 to match the reference rasterizer In this case, the Lo to edge 0 and edge 3 will be the same Since the alpha for each edge is multiplied together, we get:

edge0alpha * edge1alpha = 148/255 = 0.580392157

Since edge0alpha = edge3alpha we get:

$(edge0alpha)^2 = 0.580392157$

edge0alpha = sqrt(0.580392157) = 0.761834731 at the center pixel

The desired alpha for pixel 1 = 54/255 = 0.211764706

The slope is (0.761834731 – 0.211764706) = 0.550070025

Since we are using 8 bit precision, the slope becomes

AA Coverage [EndCap] Slope = 0.55078125

The alpha value for Lo = 0 (second pixel from center) determines the bias term and is equal to

(0.211764706 – 0.550070025) = -0.338305319

With 8 bits of precision the programmed bias value

Line Stipple

Line stipple, controlled via the **Line Stipple Enable** state variable in WM_STATE, discards certain pixels that are produced by non-AA line rasterization.

The line stipple rule is specified via the following state variables programmed via 3DSTATE_LINE_STIPPLE: the 16-bit **Line Stipple Pattern** (p), **Line Stipple Repeat Count** l, and **Line Stipple Inverse Repeat Count.** Sofware must compute **Line Stipple Inverse Repeat Count** as 1.0f / **Line Stipple Repeat Count** and then converted from float to the required fixed point encoding (see 3STATE_LINE_STIPPLE).

The WM unit maintains an internal Line Stipple Counter state variable (s) The initial value of s is zero; s is incremented after production of each pixel of a line segment (pixels are produced in order, beginning at the starting point and working towards the ending point). S is reset to 0 whenever a new primitive is

processed (unless the primitive type is LINESTRIP_CONT or LINESTRIP_CONT_BF), and before every line segment in a group of independent segments (LINELIST primitive).

During the rasterization of lines, the WM unit computes:

$$b = \lfloor s/r \rfloor \bmod 16,$$

A pixel is rendered if the bth bit of p is 1, otherwise it is discarded. The bits of p are numbered with 0 being the least significant and 15 being the most significant.

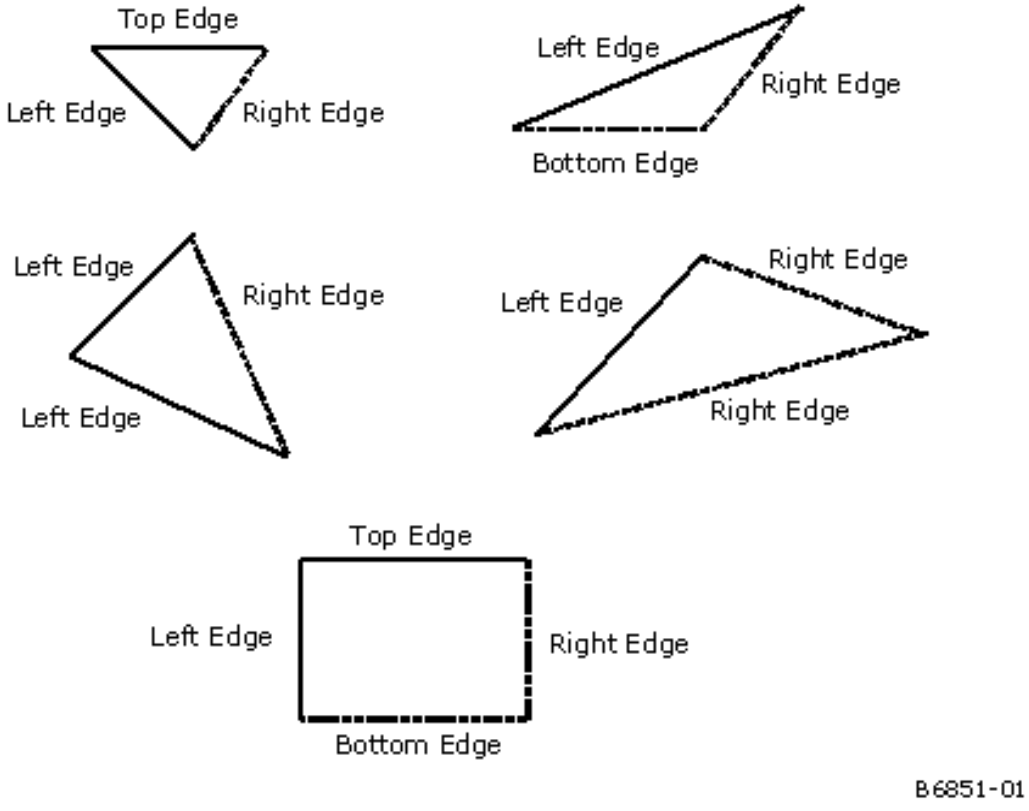
3DSTATE_LINE_STIPPLE

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



Polygon Stipple

The *Polygon Stipple* function, controlled via the **Polygon Stipple Enable** state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered Polygon stipple is applied only to the following primitive types:

3DPRIM_POLYGON
3DPRIM_TRIFAN
3DPRIM_TRILIST
3DPRIM_TRISTRIP
3DPRIM_TRISTRIP_REVERSE

Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

3DSTATE_POLY_STIPPLE_OFFSET

3DSTATE_POLY_STIPPLE_PATTERN

Multisampling

The multisampling function has two components:

- **Multisample Rasterization**: multisample rasterization occurs at a subpixel level, wherein each pixel consists of a number of "samples" at state-defined positions within the pixel footprint. Coverage of the primitive as well as color calculator operations (stencil test, depth test, color buffer blending, etc.) are done at the sample level. In addition the pixel shader itself can optionally run at the sample level depending on a separate state field.
- **Multisample Render Targets (MSRT)**: The render targets, as well as the depth and stencil buffers, now have the ability to store per-sample values. When combined with multisample rasterization, color calculator operations such as stencil test, depth test, and color buffer blending are done with the destination surface containing potentially different values per sample.

3DSTATE_MULTISAMPLE

Signal	WM_MULTISAMPLE_INT::Number of Multisamples
Description	Set the number of multisamples
Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP::Number of Multisamples : 3DSTATE_MULTISAMPLE::Number of Multisamples

3DSTATE_SAMPLE_PATTERN

Multisample ModesState

A number of state variables control the operation of the multisampling function. The following table indicates the states and their location. Refer to the state definition for more details.

State Element	Source	Description
Multisample Rasterization Mode	WM_INT::Multisample Rasterization Mode	Controls whether rasterization of non-lines is performed on a pixel or sample basis (PIXEL vs. PATTERN), and whether multisample rasterization of lines is enabled (OFF vs. ON). From this generation forward, this state element becomes an internal signal computed by other state variables (also listed here) unless certain modes are set, which can be seen in the WM_INT equation for the signal.
Multisample Dispatch Mode	WM_INT::Multisample Dispatch Mode	Controls whether the pixel shader is executed per pixel or per sample.
Number of Multisamples	3DSTATE_MULTISAMPLE and SURFACE_STATE	Indicates the number of samples per pixel contained on the surface. This field in 3DSTATE_MULTISAMPLE must match the corresponding field in SURFACE_STATE for each render target. The depth, hierarchical depth, and stencil buffers inherit this field from 3DSTATE_MULTISAMPLE.
RTIR Enabled	3DSTATE_RASTER::ForcedSampleCount != NUMRASTSAMPLES_0	Enable Render Target Independent Rasterization.
Pixel Location	3DSTATE_MULTISAMPLE	Indicates the subpixel location where values specified as “pixel” are sampled. This is either the upper left corner or the center.
MSAA Sample Offsets	3DSTATE_SAMPLE_PATTERN	For each of the N samples, specifies the subpixel location of each sample.
RTIR Sample Offsets	3DSTATE_SAMPLE_PATTERN	For each of the N samples, specifies the subpixel location of each sample.
API Mode	3DSTATE_RASTER	One of the deciding factors of what the Multisample Rasterization Mode should be according to WM_INT::Multisample Rasterization Mode. Software

State Element	Source	Description
		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	NUMRAST SAMPLES_1	NUMRAST SAMPLES_1	> NUMRAST SAMPLES_1	> NUMRAST SAMPLES_1
API Mode == DX9.0/OGL	OFF_PIXEL	OFF_PIXEL	OFF_PIXEL	ON_PATTERN	Invalid	Invalid	Invalid	Invalid
API Mode == DX10.0	OFF_PIXEL	ON_PIXEL	OFF_PIXEL	ON_PATTERN	OFF_PIXEL	Invalid	Invalid	Invalid
API Mode == DX10.1+	OFF_PIXEL	ON_PIXEL	OFF_PATTERN	ON_PATTERN	OFF_PIXEL	ON_PIXEL	OFF_PATTERN	ON_PATTERN

Definitions for lines terms used in Table 2 through Table 4:

- **Legacy Lines:** Way of drawing lines that allows Diamond Lines (SF_STATE::Line Width == 0.0), Non-anti-aliased Wide Lines (SF_STATE::Line Width != 0.0), and Line Stippling (3DSTATE_WM:: Line Stipple Enable == 1).
- **AA Lines:** Way of drawing lines that allows Anti-aliased line. These are lines rendered as rectangles that are

centered on, and aligned to, the line joining the endpoint vertices with coverage value (referred to as Anti-alias Alpha) computed per pixel.

AA Line Support Requirement
SF_INT::Anti-aliasing Enable == 1

- **MSAA Lines:** Way of drawing lines that allows Multisample Anti-aliased lines. These are lines rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices, but no Anti alias alpha coverage is computed.

Table 2: Type of Line Algorithm Given an Arrangement of State Variables

Multisample Rasterization Mode	Anti-Aliasing Enable	SF_STATE::Line Width	Line Algorithm
OFF_*	0	Non-Zero	Non-Anti-aliased Wide Lines
OFF_*	0	0.0	Diamond Lines
OFF_*	1	Non-Zero	See Note A below.
OFF_*	1	0.0	Diamond Lines
ON_*	*	*	MSAA Lines

Note A: Anti-Aliasing Details for Table 2

Anti-Aliasing Details
Anti-Aliased Lines with Alpha Coverage

Table 3: Multisample Modes with RTIR Disabled

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
NUMSAMPLES_1	OFF_PIXEL	PERSAMPLE	**Legacy Non-MSAA Mode** 1X rasterization, using Pixel Location Legacy lines or AA-line rasterization 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PIXEL	PERSAMPLE	**1X Multisampling Mode** 1X rasterization, using Pixel Location MSAA lines only, using Pixel Location 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	-	PERPIXEL	Treated the same as PERSAMPLE

	ON_PATTERN	-	Invalid
	OFF_PATTERN	-	Invalid
n where n > NUMSAMPLES_1	OFF_PIXEL	PERPIXEL	**MSRT Only, PerPixel PS** 1X rasterization, using Pixel Location See Note B below. 1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location
		PERSAMPLE	**MSRT Only, PerSample PS** 1X rasterization, using Pixel Location See Note B below. nX PS, all samples at Pixel Location nX output merge, eval Depth at Pixel Location
	ON_PIXEL	PERPIXEL	**Multibuffering MSAA, PerPixel PS** 1X rasterization, using Pixel Location MSAA lines only 1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location
		PERSAMPLE	**Multibuffering MSAA, PerSample PS** 1X rasterization, using Pixel Location MSAA lines only nX PS, all samples at Pixel Location nX output merge, eval Depth at Pixel Location
	OFF_PATTERN	PERPIXEL	**Mixed Mode, PerPixel PS** See Note B below. Non-Lines: nX rasterization, using Sample Offsets 1X PS, sample at Pixel Location nX output merge, eval depth at Sample Offsets
		PERSAMPLE	**Mixed Mode, PerSample PS** See Note B below. Non-Lines: nX rasterization, using Sample Offsets

			nX PS, sample at Pixel Location or Sample Offsets nX output merge, eval depth at Sample Offsets
	ON_PATTERN	PERPIXEL	**Pattern MSAA, PerPixel PS** nX rasterization, using Sample Offsets MSAA lines only 1X PS, sample at Pixel Location nX output merge, eval depth at Sample Offsets
		PERSAMPLE	**Pattern MSAA, PerSample PS** nX rasterization, using Sample Offsets MSAA lines only nX PS, sample at Pixel Location or Sample Offsets nX output merge, eval depth at Sample Offsets

Note B: Line Details for Table 3 and Table 4

Line Details
Legacy lines or AA-line rasterization. For PERPIXEL or PERSAMPLE in Table 3 use pixel location. For OFF_PATTERN in Table 4 use pixel location.

Table 4: Multisample Modes with RTIR Enabled

Rast Number of Samples	MS RAST MODE	HW Mode
NUMRASTSAMPLES_1	OFF_PIXEL	**Legacy Non-MSAA Mode** 1X rasterization, using Pixel Location Legacy lines or AA-line rasterization 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PIXEL	**1X Multisampling Mode** 1X rasterization, using Pixel Location MSAA lines only, using Pixel Location 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PATTERN	Invalid

	OFF_PATTERN	Invalid
n where n > NUMRASTSAMPLES_1	OFF_PIXEL	**Invalid**
	ON_PIXEL	**Invalid**
	OFF_PATTERN	**Mixed Mode, PerPixel PS** See Note B above. Non-Lines: nX rasterization, using Sample Offsets 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

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.

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: For , 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	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/2)/8) *8 * Z_Depth	see below
SURFTYPE_2D	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/2)/8) *8 * Z_Depth	see below
SURFTYPE_3D			not applicable
SURFTYPE_CUBE	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/2)/8) *8 * 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=8. The following equation gives the minimum HZ_QPitch based on largest LOD m defined in the surface:

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

If m is less than 2, treat all h_L with L > m as zero and use the above equation.

The minimum HZ_Height required for a 3D surface must be computed based on h_L parameters documented in the GPU Overview volume, and the maximum LOD m:

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 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 Workaround

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. DepthStall and DepthFlush are not needed between consecutive depth clear passes nor is it required if the depth-clear pass was done with "full_surf_clear" bit set in the 3DSTATE_WM_HZ_OP.

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).

Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete If the depth buffer is retained and used for another purpose (i.e as input to the sampling engine as a shadow map), it must first be "resolved" This is done by setting the **Depth Buffer Resolve Enable** field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

The following is required when performing a depth buffer resolve:

- The surface must have been initialized with a Depth Buffer Clear after its allocation to initialize the Depth Clear Value.
- A rectangle primitive of the same size as the previous depth buffer clear operation must be delivered, and depth buffer state cannot have changed since the previous depth buffer clear operation.
- **Depth Test Enable** must be enabled with the **Depth Test Function** set to NEVER. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

Programming Note	
Context:	HTML
HW uses the clear value from the 3DSTATE_CLEAR_PARAM. If you change the value in the 3DSTATE_CLEAR_PARAMS before resolve, it will flush the depth caches and have the new-clear value in its register. When doing the resolve pass, it is driver's responsibility to make sure that the clear-value for the depth buffer is the same one as the clear-pass.	

Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation If this occurs, the hierarchical depth buffer must be *resolved* to avoid incorrect device behavior. This is done by setting the Hierarchical Depth Buffer Resolve Enable field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be delivered.
- **Depth Test Enable** must be disabled. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel**, and **Pixel Shader Computed Depth** must all be disabled.

Optimized Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete If the depth buffer is retained and used for another purpose (i.e locked by the app), it must first be "resolved" This is done by setting the **Depth Buffer Resolve Enable** field in 3DSTATE_WM_HZ_OP. The depth buffer resolve uses the same sequence as the optimized Depth buffer clear (see above) except the **Depth Buffer Resolve Enable** bit is set. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

Doing a resolve operation requires that a preceding Depth Buffer Clear operation is required to have initialized the Depth Clear Value.

Optimized Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation. If this occurs, the hierarchical depth

buffer must be "resolved" to avoid incorrect device behavior. This is done by setting the **Hierarchical Depth Buffer Resolve Enable** field in 3DSTATE_WM_HZ_OP and specifying a full render target sized rectangle. The depth buffer resolve uses the same sequence as the optimized Depth buffer clear (see above) except the **Hierarchical Depth Buffer Resolve Enable** bit is set. Once this is complete, the hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be programmed on Xmin, Ymin, Xmax, and Ymax in the 3DSTATE_WM_HZ_OP command.
- The rectangle primitive size must be aligned to 8x4 pixels.

Separate Stencil Buffer

The following tables describe the separate stencil buffer for different generations.

The separate stencil buffer is always enabled, thus the field in 3DSTATE_DEPTH_BUFFER to explicitly enable the separate stencil buffer has been removed. Surface formats with interleaved depth and stencil are no longer supported.

The stencil buffer has a format of R8_UNIT, and shares **Surface Type**, **Height**, **Width**, and **Depth**, **Minimum Array Element**, **Render Target View Extent**, **Depth Coordinate Offset X/Y**, **LOD**, and **Depth Buffer Object Control State** fields of the depth buffer.

DepthStencil Buffer State

This section contains the state registers for the Depth/Stencil Buffers.

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.

For each fragment, the Pixel Shader calculates the values of the various vertex attributes that are to be interpolated across the object using the interpolation coefficients. It then executes an API-supplied Pixel Shader Program. Instructions in this program permit the accessing of texture map data, where Texture Samplers are employed to sample and filter texture maps (see the Shared Functions chapter). Arithmetic operations can be performed on the texture data, input fragment information, and Pixel Shader Constants to compute the resultant fragment's output. The Pixel Shader program also allows the pixel to be discarded from further processing.

3DSTATE_PS

This command is used to set state used by the pixel shader dispatch stage.

Command
3DSTATE_PS

Programming Note	
Context:	Pixel Shader Thread Generation
Note: The PS Unit also receives 3DSTATE_PS_BLEND, 3DSTATE_SAMPLEMASK, 3DSTATE_MULTISAMPLE, and 3DSTATE_PS_EXTRA.	

Signal	PS_INT::oMask Present to RenderTarget
Description	This bit is inserted in the PS payload header and made available to the DataPort (either via the message header or via header bypass) to indicate that oMask data (one or two phases) is included in Render Target Write messages. If present, the oMask data is used to mask off samples.
Formula	= 3DSTATE_PS_EXTRA::oMask Present to RenderTarget

Signal	PS_INT::Dual Source Blend Enable
Description	This field is set if dual source blend is enabled. If this bit is disabled, the data port dual source message reverts to a single source message using source 0.
Formula	= 3DSTATE_PS_BLEND::ColorBufferBlendEnable && (PS_INT::UsesSrc1BlendFactor || (PS_INT::IndependentAlphaUsesSrc1BlendFactors && 3DSTATE_PS_BLEND::Independent Alpha Blend Enable))

Signal	PS_INT::UsesSrc1BlendFactor
Description	
Formula	= (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA)

Signal	PS_INT::IndependentAlphaUsesSrc1BlendFactors
Description	
Formula	= (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA)

Signal	PS_INT::PS UAV-only
Description	Pixel Shader UAV-only render target
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Has UAV && !3DSTATE_PS_EXTRA:: Pixel Shader Does not write to RT

Command
3DSTATE_PS_EXTRA This command is used to set state used by the pixel shader dispatch stage
3DSTATE_PS_BLEND This command is used to set state used by the pixel shader dispatch stage
3DSTATE_CONSTANT_PS

Command
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.

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	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.

The driver must ensure that the PS kernel begins with a corresponding jump table to properly handle the number of subspans dispatched. The WM unit will "OR" in the two LSBs of the Kernel Pointer (bits 5:4) to create an instruction level address. (Note that the pointer from WM_STATE is 64-byte aligned which corresponds to four 128-bit instructions.)

If only one dispatch mode is enabled, the Jitter should not include any jump table entries at the beginning of the PS kernel. If multiple dispatch modes are enabled, a two entry jump table should always be inserted, regardless of which modes are enabled (jump table entry for 8 pixel dispatch, followed by jump table entry for 32 pixel dispatch).

Note that for SIMD32 dispatch, pixel shader dispatch function increments GRF Start Register for URB Data state by 2 to account for the additional SIMD16 payload. The Pixel Shader kernel needs to comprehend this modification for SIMD32.

```
if ( 32PixelDispatchEnable && n > 7 )
    Dispatch 32 Pixels
else if ( 16PixelDispatchEnable && ( n > 2 || ! 8PixelDispatchEnable) )
    Dispatch 16 Pixels
else
    Dispatch 8 Pixels
```

```
end if
```

Multisampling Effects on Pixel Shader Dispatch

The pixel shader payloads are defined in terms of subspans and pixels. The slots in the pixel shader thread previously mapped 1:1 with pixels. With multisampling, a slot could contain a pixel or may just contain a single sample, depending on the mode. Payload definitions now refer to *slot* to make the definition independent of multisampling mode.

MSDISPMODE_PERPIXEL Thread Dispatch

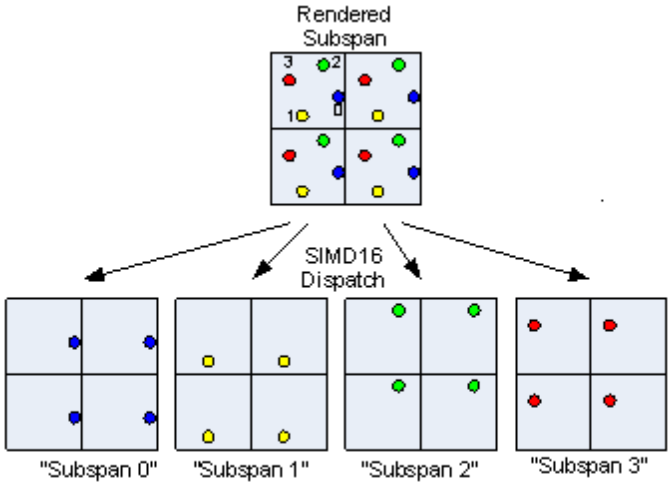
In PERPIXEL mode, the pixel shader kernel still works on 2/4/8 separate subspans, depending on dispatch mode. The fact that rasterization and the depth/stencil tests are being performed on a per-sample (not per-pixel) basis is transparent to the pixel shader kernel.

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

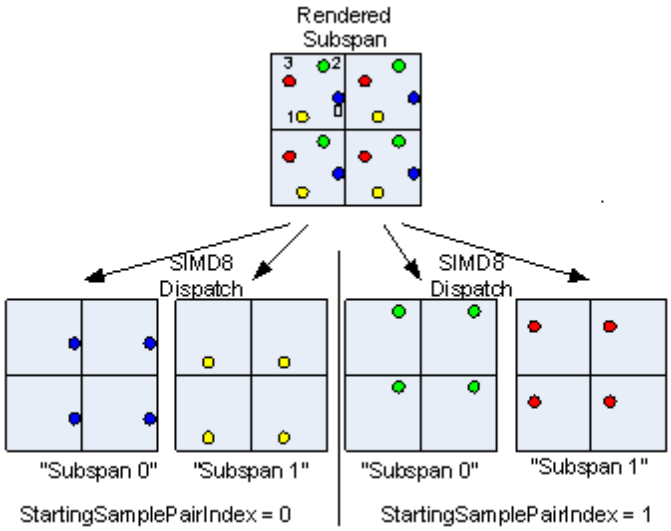
MSDISPMODE_PERSAMPLE Thread Dispatch

In PERSAMPLE mode, the pixel shader needs to operate on a sample vs. pixel basis (although this collapses in NUMSAMPLES_1 mode) Instead of processing strictly different subspans in parallel , the PS kernel processes different sample indices of one or more subspans in parallel For example, a SIMD16 dispatch in PERSAMPLE/NUMSAMPLES_4 mode would operate on a single subspan, with the usual "4 Subspan0 pixel slots" used for the "4 Sample0 locations of the (single) subspan" Subspan1 slots would be used for the Sample1 locations, and so on This layout allows the pixel shader to compute derivatives/LOD based on deltas between corresponding sample locations in the subspan in the same fashion as LEGACY pixel shader execution, and as required by DX10.1.

Depending on the dispatch mode (8/16/32 pixels) and multisampling mode (1X/4X), there are different mappings of subspans/samples onto dispatches and slots-within-dispatch In some cases, more than one subspan may be included in a dispatch, while in other cases multiple dispatches are be required to process all samples for a single subspan In the latter case, the **StartingSamplePairIndex** value is included in the payload header so the Render Target Write message will access the correct samples with each message.



PERSAMPLE SIMD16 4X Dispatch



PERSAMPLE Dispatch

The following table provides the complete dispatch/slot mappings for all the MS/Dispatch combinations.

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
SIMD32	*1X*	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0] *Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0]* *Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0]* *Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]*
	2X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]* *Slot[19:16] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[23:20] = Subspan[2].Pixel[3:0].Sample[1]* *Slot[27:24] = Subspan[3].Pixel[3:0].Sample[0]* *Slot[31:28] = Subspan[3].Pixel[3:0].Sample[1]*
	4X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]* *Slot[19:16] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[23:20] = Subspan[1].Pixel[3:0].Sample[1]* *Slot[27:24] = Subspan[1].Pixel[3:0].Sample[2]* *Slot[31:28] = Subspan[1].Pixel[3:0].Sample[3]*
	8X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]* *Slot[19:16] = Subspan[0].Pixel[3:0].Sample[4]* *Slot[23:20] = Subspan[0].Pixel[3:0].Sample[5]* *Slot[27:24] = Subspan[0].Pixel[3:0].Sample[6]* *Slot[31:28] = Subspan[0].Pixel[3:0].Sample[7]*
	16x	*Dispatch[i]: (i=0, 4)*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		SSPI = i *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]* *Slot[19:16] = Subspan[0].Pixel[3:0].Sample[SSPI*2+4]* *Slot[23:20] = Subspan[0].Pixel[3:0].Sample[SSPI*2+5]* *Slot[27:24] = Subspan[0].Pixel[3:0].Sample[SSPI*2+6]* *Slot[31:28] = Subspan[0].Pixel[3:0].Sample[SSPI*2+7]*
SIMD16	*1X*	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]*
	2X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]*
	4X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]*
	8X	*Dispatch[i]: (i=0, 2)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]*
	16x	*Dispatch[i]: (i=0, 2, 4, 6)*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		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 thread count which is incremented by the Windower for every 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 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 thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the Sampler State table. It is specified as an offset from the **Dynamic State Base Address**. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>thread.

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

Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two</td></tr>
<tr><td>R0.2</td><td>31:0</td><td>Reserved: Delivered as zeros (reserved for message header fields).</td></tr>
<tr><td rowspan="2">R0.1</td><td>31:6</td><td>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.

Format = DynamicStateOffset[31:5]</td></tr>
<tr><td>5:0</td><td>Reserved</td></tr>
<tr><td rowspan="4">R0.0</td><td>31</td><td>Reserved</td></tr>
<tr><td>30:27</td><td>Viewport Index: Specifies the index of the viewport currently being used.

Format = U4

Range = [0,15]</td></tr>
<tr><td>26:16</td><td>Render Target Array Index: Specifies the array index to be used for the following surface types:

SURFTYPE_1D: specifies the array index Range = [0,2047]

SURFTYPE_2D: specifies the array index Range = [0,2047]

SURFTYPE_3D: specifies the “r” coordinate Range = [0,2047]

SURFTYPE_CUBE: specifies the face identifier Range = [0,5]

<table><tr><th>Face</th><th>Render Target Array Index</th></tr><tr><td>+x</td><td>0</td></tr><tr><td>-x</td><td>1</td></tr><tr><td>+y</td><td>2</td></tr><tr><td>-y</td><td>3</td></tr><tr><td>+z</td><td>4</td></tr><tr><td>-z</td><td>5</td></tr></table>
Format = U11</td></tr>
<tr><td>15</td><td>Front/Back Facing Polygon: Determines whether the polygon is front or back facing. Used by</td></tr>
</table>

DWord	Bits	Description
		the render cache to determine which stencil test state to use. 0: Front Facing 1: Back Facing
	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.
	11:9	Reserved
	8	Reserved for expansion of **Starting Sample Pair Index**.
	7:6	**Starting Sample Pair Index:** Indicates the index of the first sample pair of the dispatch. Format = U2 Range = [0,3]
	5	Reserved
	4:0	**Primitive Topology Type:** This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST). Format: (See 3DPRIMITIVE command in *3D Pipeline*.)
R1.7	31:16	**Pixel/Sample Mask (SubSpan[3:0]):** Indicates which pixels within the four subspans are lit. If 32 pixel dispatch is enabled, this field contains the pixel mask for the first four subspans. **Note:** This is not a duplicate of the Dispatch Mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel/Sample Mask Copy (SubSpan[3:0]):** This is a duplicate copy of the pixel mask. This copy can be modified as the pixel shader thread executes in order to turn off pixels based on kill instructions.
R1.6	31:0	Reserved
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

DWord	Bits	Description
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
R1.0	31:20	Reserved
	15:12	Slot 3 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	11:8	Slot 2 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	7:4	Slot 1 SampleID (if pixel or sample dispatch) Format = U4

DWord	Bits	Description
		1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	3:0	Slot 0 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
		R2: Delivered only if this is a *32-pixel dispatch*.
R2.7	31:16	**Pixel/Sample Mask (SubSpan[7:4]):** Indicates which pixels within the upper four subspans are lit. This field is valid only when the 32 pixel dispatch state is enabled. This field must not be modified by the pixel shader thread. Note: This is not a duplicate of the dispatch mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel/Sample Mask Copy (SubSpan[7:4]):** This is a duplicate copy of pixel mask for the upper 16 pixels. This copy will be modified as the pixel shader thread executes to turn off pixels based on kill instructions.
R2.6	31:0	Reserved
R2.5	31:16	**Y7:** Y coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16
	15:0	**X7:** X coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16
R2.4	31:16	Y6
	15:0	X6
R2.3	31:16	Y5
	15:0	X5
R2.2	31:16	Y4

DWord	Bits	Description
	15:0	X4
R2.1	31:0	Reserved
R2.0	31:16	Reserved
	15:12	Slot 7 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	11:8	Slot 6 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	7:4	Slot 5 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	3:0	Slot 4 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]

DWord	Bits	Description
		16X MSAA range [0..15]
		R3-R26: Delivered only if the corresponding **Barycentric Interpolation Mode** bit is set. Register phases containing Slot 8-15 data are not delivered in *8-pixel dispatch* mode.
R3.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 7 This and the next register phase is only included if the corresponding enable bit in **Barycentric Interpolation Mode** is set. Format = IEEE_Float
R3.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 6
R3.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 5
R3.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 4
R3.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 3
R3.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 2
R3.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 1
R3.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 0
R4		Perspective Pixel Location Barycentric[2] for Slots 7:0
R5.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 15
R5.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 14
R5.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 13
R5.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 12
R5.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 11
R5.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 10
R5.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 9
R5.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 8
R6		Perspective Pixel Location Barycentric[2] for Slots 15:8
R7:10		Perspective Centroid Barycentric
R11:14		Perspective Sample Barycentric
R15:18		Linear Pixel Location Barycentric
R19:22		Linear Centroid Barycentric
R23:26		Linear Sample Barycentric
		R27: Delivered only if **Pixel Shader Uses Source Depth** is set.
R27.7	31:0	Interpolated Depth for Slot 7 Format = IEEE_Float This and the next register phase is only included if **Pixel Shader Uses Source Depth**

DWord	Bits	Description
		(WM_STATE) is set.
R27.6	31:0	Interpolated Depth for Slot 6
R27.5	31:0	Interpolated Depth for Slot 5
R27.4	31:0	Interpolated Depth for Slot 4
R27.3	31:0	Interpolated Depth for Slot 3
R27.2	31:0	Interpolated Depth for Slot 2
R27.1	31:0	Interpolated Depth for Slot 1
R27.0	31:0	Interpolated Depth for Slot 0
		R28: Delivered only if **Pixel Shader Uses Source Depth** is set and this is not an *8-pixel dispatch*.
R28.7	31:0	Interpolated Depth for Slot 15
R28.6	31:0	Interpolated Depth for Slot 14
R28.5	31:0	Interpolated Depth for Slot 13
R28.4	31:0	Interpolated Depth for Slot 12
R28.3	31:0	Interpolated Depth for Slot 11
R28.2	31:0	Interpolated Depth for Slot 10
R28.1	31:0	Interpolated Depth for Slot 9
R28.0	31:0	Interpolated Depth for Slot 8
		R29: Delivered only if **Pixel Shader Uses Source W** is set.
R29.7	31:0	Interpolated W for Slot 7 Format = IEEE_Float This and the next register phase are only included if **Pixel Shader Uses Source W** (WM_STATE) is set.
R29.6	31:0	Interpolated W for Slot 6
R29.5	31:0	Interpolated W for Slot 5
R29.4	31:0	Interpolated W for Slot 4
R29.3	31:0	Interpolated W for Slot 3
R29.2	31:0	Interpolated W for Slot 2
R29.1	31:0	Interpolated W for Slot 1
R29.0	31:0	Interpolated W for Slot 0
		R30: Delivered only if **Pixel Shader Uses Source W** is set and this is not an *8-pixel dispatch*.
R30.7	31:0	Interpolated W for Slot 15
R30.6	31:0	Interpolated W for Slot 14
R30.5	31:0	Interpolated W for Slot 13

DWord	Bits	Description
R30.4	31:0	Interpolated W for Slot 12
R30.3	31:0	Interpolated W for Slot 11
R30.2	31:0	Interpolated W for Slot 10
R30.1	31:0	Interpolated W for Slot 9
R30.0	31:0	Interpolated W for Slot 8
		R31: Delivered only if **Position XY Offset Select** is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE.
R31.7	31:24	Position Offset Y for Slot 15 This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of **Position XY Offset Select**. Format = U4.4 For non-CP rate dispatch: Range = [0.0,1.0) For CP rate dispatch: Range = [0.0,4.0)
	23:16	Position Offset X for Slot 15 This field contains either the CENTROID or SAMPLE position offset for X, depending on the state of **Position XY Offset Select**. Format = U4.4 For non-CP rate dispatch: Range = [0.0,1.0) For CP rate dispatch: Range = [0.0,4.0)
	15:8	Position Offset Y for Slot 14
	7:0	Position Offset X for Slot 14
R31.6	31:24	Position Offset Y for Slot 13
	23:16	Position Offset X for Slot 13
	15:8	Position Offset Y for Slot 12
	7:0	Position Offset X for Slot 12
R31.5:4		Position Offset X/Y for Slot[11:8]
R31.3:2		Position Offset X/Y for Slot[7:4]
R31.1:0		Position Offset X/Y for Slot[3:0]
		R32: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set.
R32.7	31:0	Input Coverage Mask for Slot 7 Format = U32 This and the next register phase is only included if **Pixel Shader Uses Input Coverage Mask**

DWord	Bits	Description
		(3DSTATE_PS) is set. This field always encodes sample Coverage Mask.
R32.6	31:0	Input Coverage Mask for Slot 6
R32.5	31:0	Input Coverage Mask for Slot 5
R32.4	31:0	Input Coverage Mask for Slot 4
R32.3	31:0	Input Coverage Mask for Slot 3
R32.2	31:0	Input Coverage Mask for Slot 2
R32.1	31:0	Input Coverage Mask for Slot 1
R32.0	31:0	Input Coverage Mask for Slot 0
		R33: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set and this is not an *8-pixel dispatch*.
R33.7	31:0	Input Coverage Mask for Slot 15
R33.6	31:0	Input Coverage Mask for Slot 14
R33.5	31:0	Input Coverage Mask for Slot 13
R33.4	31:0	Input Coverage Mask for Slot 12
R33.3	31:0	Input Coverage Mask for Slot 11
R33.2	31:0	Input Coverage Mask for Slot 10
R33.1	31:0	Input Coverage Mask for Slot 9
R33.0	31:0	Input Coverage Mask for Slot 8
		R34-R57: Delivered only if the corresponding **Barycentric Interpolation Mode** bit is set and this is a *32-pixel dispatch*.
R34.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 23 This and the next register phase is only included if the corresponding enable bit in **Barycentric Interpolation Mode** is set. Format = IEEE_Float
R34.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 22
R34.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 21
R34.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 20
R34.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 19
R34.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 18
R34.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 17
R34.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 16
R35		Perspective Pixel Location Barycentric[2] for Slots 23:16
R36.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 31

DWord	Bits	Description
R36.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 30
R36.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 29
R36.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 28
R36.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 27
R36.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 26
R36.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 25
R36.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 24
R37		Perspective Pixel Location Barycentric[2] for Slots 31:24
R38:41		Perspective Centroid Barycentric
R42:45		Perspective Sample Barycentric
R46:49		Linear Pixel Location Barycentric
R50:53		Linear Centroid Barycentric
R54:57		Linear Sample Barycentric
		R58-R59: Delivered only if **Pixel Shader Uses Source Depth** is set and this is a *32-pixel dispatch*.
R58.7	31:0	Interpolated Depth for Slot 23 Format = IEEE_Float This and the next register phase is only included if **Pixel Shader Uses Source Depth** (WM_STATE) bit is set.
R58.6	31:0	Interpolated Depth for Slot 22
R58.5	31:0	Interpolated Depth for Slot 21
R58.4	31:0	Interpolated Depth for Slot 20
R58.3	31:0	Interpolated Depth for Slot 19
R58.2	31:0	Interpolated Depth for Slot 18
R58.1	31:0	Interpolated Depth for Slot 17
R58.0	31:0	Interpolated Depth for Slot 16
R59.7	31:0	Interpolated Depth for Slot 31
R59.6	31:0	Interpolated Depth for Slot 30
R59.5	31:0	Interpolated Depth for Slot 29
R59.4	31:0	Interpolated Depth for Slot 28
R59.3	31:0	Interpolated Depth for Slot 27
R59.2	31:0	Interpolated Depth for Slot 26
R59.1	31:0	Interpolated Depth for Slot 25
R59.0	31:0	Interpolated Depth for Slot 24
		R60-R61: Delivered only if **Pixel Shader Uses Source W** is set and this is a *32-pixel dispatch*.

DWord	Bits	Description
R60.7	31:0	Interpolated W for Slot 23 Format = IEEE_Float This and the next register phase are only included if **Pixel Shader Uses Source W** (WM_STATE) bit is set.
R60.6	31:0	Interpolated W for Slot 22
R60.5	31:0	Interpolated W for Slot 21
R60.4	31:0	Interpolated W for Slot 20
R60.3	31:0	Interpolated W for Slot 19
R60.2	31:0	Interpolated W for Slot 18
R60.1	31:0	Interpolated W for Slot 17
R60.0	31:0	Interpolated W for Slot 16
R61.7	31:0	Interpolated W for Slot 31
R61.6	31:0	Interpolated W for Slot 30
R61.5	31:0	Interpolated W for Slot 29
R61.4	31:0	Interpolated W for Slot 28
R61.3	31:0	Interpolated W for Slot 27
R61.2	31:0	Interpolated W for Slot 26
R61.1	31:0	Interpolated W for Slot 25
R61.0	31:0	Interpolated W for Slot 24
		R62: Delivered only if **Position XY Offset Select** is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE and this is a *32-pixel dispatch*.
R62.7	31:24	Position Offset Y for Slot 31 This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of **Position XY Offset Select**. Format = U4.4 Range = [0.0,1.0)
	23:16	Position Offset X for Slot 31 This field contains either the CENTROID or SAMPLE position offset for X, depending on the state of **Position XY Offset Select**. Format = U4.4 Range = [0.0,1.0)
	15:8	Position Offset Y for Slot 30
	7:0	Position Offset X for Slot 30

DWord	Bits	Description
R62.6	31:24	Position Offset Y for Slot 29
	23:16	Position Offset X for Slot 29
	15:8	Position Offset Y for Slot 28
	7:0	Position Offset X for Slot 28
R62.5:4		Position Offset X/Y for Slot[27:24]
R62.3:2		Position Offset X/Y for Slot[23:20]
R62.1:0		Position Offset X/Y for Slot[19:16]
		R63-R64: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set and this is a *32-pixel dispatch*.
R63.7	31:0	Input Coverage Mask for Slot 23 Format = U32 This and the next register phase are only included if **Pixel Shader Uses Input Coverage Mask** (3DSTATE_PS) is set.
R63.6	31:0	Input Coverage Mask for Slot 22
R63.5	31:0	Input Coverage Mask for Slot 21
R63.4	31:0	Input Coverage Mask for Slot 20
R63.3	31:0	Input Coverage Mask for Slot 19
R63.2	31:0	Input Coverage Mask for Slot 18
R63.1	31:0	Input Coverage Mask for Slot 17
R63.0	31:0	Input Coverage Mask for Slot 16
R64.7	31:0	Input Coverage Mask for Slot 31
R64.6	31:0	Input Coverage Mask for Slot 30
R64.5	31:0	Input Coverage Mask for Slot 29
R64.4	31:0	Input Coverage Mask for Slot 28
R64.3	31:0	Input Coverage Mask for Slot 27
R64.2	31:0	Input Coverage Mask for Slot 26
R64.1	31:0	Input Coverage Mask for Slot 25
R64.0	31:0	Input Coverage Mask for Slot 24
[Varies] optional	255:0	**Constant Data (optional):** For more details about the size and source of constant data, please refer to General Programming of Thread-Generating Stages in the Push Constants chapter.

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 for SNB+ 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
• 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.

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 propogated 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.

Blending	RT Format	Pre-Blend Color Clamp	Post-Blend Color Clamp
Off	UNORM, UNORM_SRGB,YCRCB	Must be enabled with range = RT range or [0,1] (same function)	N/A, state ignored
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	N/A, state ignored
	FLOAT (except for R11G11B10_FLOAT)	Must be enabled (with any desired range)	N/A, state ignored
	R11G11B10_FLOAT	Must be enabled with either [0,1] or RT range	N/A, state ignored
	UINT, SINT	State ignored, implied clamp to RT range	N/A, state ignored
On (where permitted)	UNORM, UNORM_SRGB	Must be enabled with range = RT range or [0,1] (same function)	Must be enabled with range = RT range or [0,1] (same function)
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	Must be enabled with range = RT range or [-1,1] (same function)
	FLOAT (except for R11G11B10_FLOAT)	Can be disabled or enabled (with any desired range)	Must be enabled (with any desired range)
	R11G11B10_FLOAT	Can be disabled or enabled (with any desired range)	Must be enabled with either [0,1] or RT range

Pre-Blend Color Clamping When Blending is Disabled

The clamping of source color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all source color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed.

Programming Note	
Context:	Pre-Blend Color Clamping When Blending is Disabled

- Given the possibility of writing UNPREDICTABLE values to the Color Buffer, it is expected and highly recommended that, when blending is disabled, software set **Pre-Blend Color Clamp Enable** to ENABLED and select an appropriate **Color Clamp Range**.
- When using SINT or UINT rendertarget surface formats, **Blending must** be DISABLED. The **Pre-Blend Color Clamp Enable** and **Color Clamp Range** fields are ignored, and an implied clamp to the rendertarget surface format is performed.

Pre-Blend Color Clamping When Blending is Enabled

The clamping of source, destination and constant color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all these color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed on these color components prior to blending.

Color Buffer Blending

The Color Buffer Blending function is used to combine one or two incoming "source" pixel color+alpha values with the "destination" color+alpha read from the corresponding location in a RenderTarget.

Blending is enabled on a global basis by the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE). If DISABLED, Blending and Post-Blend Clamp functions are disabled for all RenderTargets, and the pixel values (possibly subject to Pre-Blend Clamp) are passed through unchanged.

The Color Buffer Blend Enable is in the per-render-target BLEND_STATE, and the field in SURFACE_STATE is no longer supported.

Programming Note	
Context:	Color Buffer Blending, Logic Ops, DataPort, surface formats, render targets

- Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
- Dual source blending: The DataPort only supports dual source blending with a SIMD8-style message.
- Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in *Sampling Engine*. Blending must be disabled on a RenderTarget if blending is not supported.

The incoming "source" pixel values are modulated by a selected "source" blend factor, and the possibly gamma-decorrected "destination" values are modulated by a "destination" blend factor. These terms are then combined with a "blend function". In general:

src_term = src_blend_factor * src_color

dst_term = dst_blend_factor * dst_color

color output = blend_function(src_term, dst_term)

If there is no alpha value contained in the Color Buffer, a default value of 1.0 is used and, correspondingly, there is no alpha component computed by this function.

Dual Source Blending: When using "Dual Source" Render Target Write messages, the Source1 pixel color+alpha passed in the message can be selected as a src/dst blend factor. See Color Buffer Blend Color Factors. In single-source mode, those blend factor selections are invalid. If SRC1 is included in a src/dst blend factor and a DualSource RT Write message is not used, results are UNDEFINED. (This reflects the same restriction in DX APIs, where undefined results are produced if "o1" is not written by a PS – there are no default values defined). If SRC1 is not included in a src/dst blend factor, dual source blending must be disabled.

The blending of the color and alpha components is controlled with two separate (color and alpha) sets of state variables. However, if the **Independent Alpha Blend Enable** state variable in COLOR_CALC_STATE is DISABLED, then the "color" (rather than "alpha") set of state variables is used for both color and alpha. Note that this is the only use of the **Independent Alpha Blend Enable** state – it does not control whether Blending occurs, only how.

Per **Render Target Blend State**: Blend state is selected based on **Render Target Index** contained in the message header, and appropriate blend state is applied to Render Target Write messages.

The following table describes the color source and destination blend factors controlled by the **Source [Alpha] Blend Factor** and **Destination [Alpha] Blend Factor** state variables in COLOR_CALC_STATE. Note that the blend factors applied to the R,G,B channels are always controlled by the **Source/Destination Blend Factor**, while the blend factor applied to the alpha channel is controlled either by **Source/Destination Blend Factor** or **Source/Destination Alpha Blend Factor**.

Color Buffer Blend Color Factors

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_SRC1_COLOR	o1.r, o1.g, o1.b, o1.a

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_INV_SRC1_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) **Programming Note:** This is a superset of the OpenGL "max" function.

Post-Blend Color Clamping

(See *Pre-Blend Color Clamping* above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

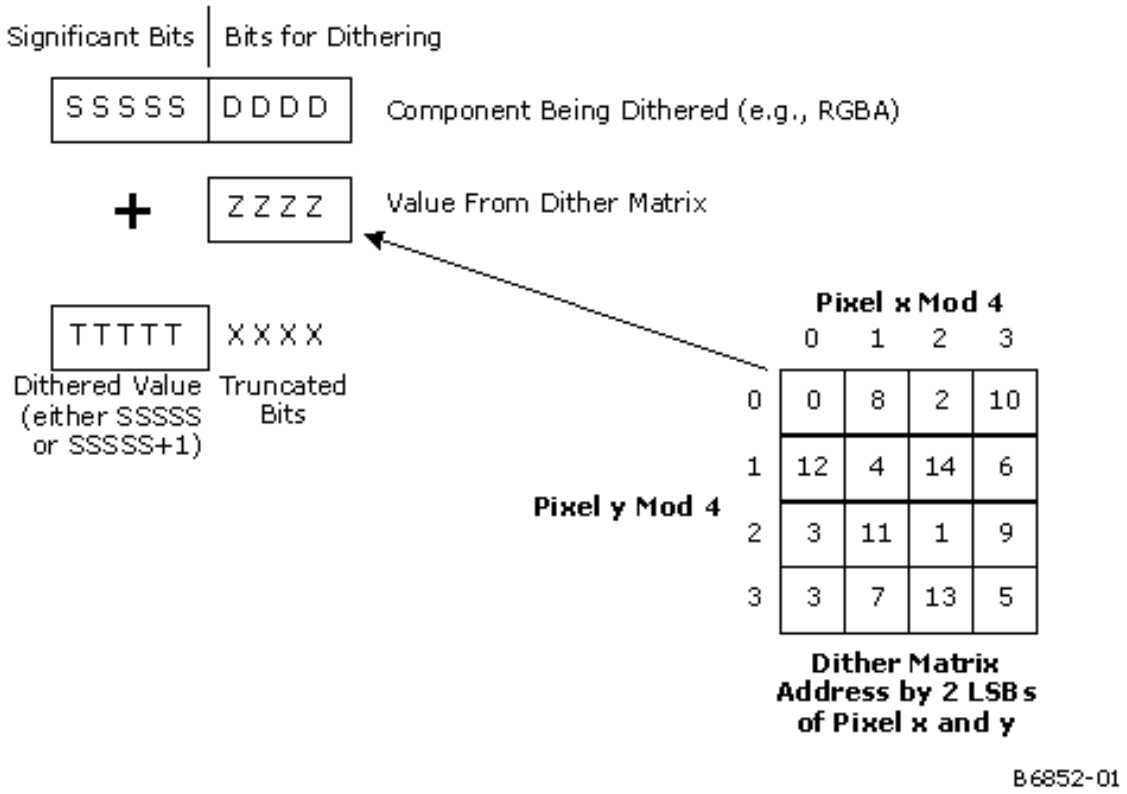
Regardless of the setting of **Post-Blend Color Clamp Enable**, when Blending is enabled color components will be automatically clamped to (at least) the rendertarget surface format range at this stage of the pipeline.

Dithering

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer's eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel's X and Y screen coordinate. The pixel's X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel's screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel's screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.



Dithering Process (5-Bit Example)

Logic Ops

The Logic Ops function is used to combine the incoming "source" pixel color/alpha values with the corresponding "destination" color/alpha contained in the ColorBuffer, using a logic function.

The Logic Op function is enabled by the **LogicOp Enable** state variable. If DISABLED, this function is ignored and the incoming pixel values are passed through unchanged.

Programming Notes

Programming Note
Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
Logic Ops are supported on all blendable render targets and render targets with *INT formats.

The following table lists the supported logic ops. The logic op is selected using the **Logic Op Function** field in COLOR_CALC_STATE.

Logic Ops

LogicOp Function	Definition (S=Source, D=Destination)
LOGICOP_CLEAR	all 0's
LOGICOP_NOR	NOT (S OR D)

LogicOp Function	Definition (S=Source, D=Destination)
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

Stencil Operation	Description
STENCILOP_REPLACE	Store the *StencilTestReference* reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible wrap around to 0)
STENCILOP_DECR	Decrement (possible wrap to max value)
STENCILOP_INVERT	Logically invert the stencil value

Any and all writes to the stencil portion of the depth buffer are enabled by the **Stencil Buffer Write Enable** state variable.

When writes are enabled, the **Stencil Buffer Write Mask** and **Backface Stencil Buffer Write Mask** state variables provide an 8-bit mask that selects which bits of the stencil write value are modified. Masked-off bits (i.e., mask bit == 0) are left unmodified in the Stencil Buffer.

Programming Note	
Context:	Stencil Buffer Updates
The Stencil Buffer can be written even if depth buffer writes are disabled via Depth Buffer Write Enable	

Depth Buffer Updates

Any and all writes to the Depth Buffer are enabled by the **Depth Buffer Write Enable** state variable. If there is no Depth Buffer, writes must be explicitly disabled with this state variable, or operation is UNDEFINED.

If depth testing is disabled or the depth test passed, the incoming pixel's depth value is written to the Depth Buffer. If depth testing is enabled and the depth test failed, the pixel is discarded – with no modification to the Depth or Color Buffers (though the Stencil Buffer may have been modified).

Color Gamma Correction

Computed RGB (not A) channels can be gamma-corrected prior to update of the Color Buffer.

This function is automatically invoked whenever the destination surface (render target) has an SRGB format (see surface formats in *Sampling Engine*). For these surfaces, the computed RGB values are converted from gamma=1.0 space to gamma=2.4 space by applying a ^(2.4) exponential function.

Color Buffer Updates

Finally, if the pixel has not been discarded by this point, the incoming pixel color is written into the Color Buffer. The **Surface Format** of the color buffer indicates which channel(s) are written (e.g., R8G8_UNORM are written with the Red and Green channels only). The **Color Buffer Component Write Disables** from the Color Buffer's SURFACE_STATE provide an independent write disable for each channel of the Color Buffer.

Pixel Pipeline State Summary

COLOR_CALC_STATE

3DSTATE_BLEND_STATE_POINTERS

3DSTATE_BLEND_STATE_POINTERS

3DSTATE_DEPTH_STENCIL_STATE_POINTERS

3DSTATE_DEPTH_STENCIL_STATE_POINTERS does not exist for BDW+. 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 on BDW+. 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.

MCS Buffer 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.

MSAA	Width of Clear Rect	Height of Clear Rect
2x	Ceil(1/8*width)	Ceil(1/2*height)
4X	Ceil(1/8*width)	Ceil(1/2*height)
8X	Ceil(1/2*width)	Ceil(1/2*height)
16X	width	Ceil(1/2*height)

- **MSAA Compression:** Multi-sample render target is bound to the pipeline and MSAA compression feature is enabled. In this case, MCS buffer stores the information required for MSAA compression algorithm. The size and layout of the MCS buffer is based on per-pixel RT. For 4X and 8X MSAA, MCS buffer element is 8bpp and 32bpp respectively. Height, width, and layout of MCS buffer in this case must match with Render Target height, width, and layout. MCS buffer is tiledY. When MCS buffer is enabled and bound to MSRT, it is required that it is cleared prior to any rendering. A clear value can be specified optionally in the surface state of the corresponding RT. Clear pass for this case requires that scaled down primitive is sent down with upper left coordinate to coincide with actual rectangle being cleared. For MSAA, clear rectangle's height and width need to as show in the following table in terms of (width,height) of the RT.
- **Fast Color Clear:** When non multi-sample render target is bond to the pipeline and MSC buffer is enabled, MCS buffer is used as an intermediate (coarse granular) buffer per RT. Hence, MCS buffer is used to improve render target clear. When MCS is buffer is used for color clear of non-multisampler render target, the following restrictions apply:

Color Clear of Non-MultiSampler Render Target Restrictions

<table>
<tr><th>Restrictions</th></tr>
<tr><td>Support is limited to tiled render targets.</td></tr>
<tr><td>Mip-mapped and arrayed surfaces are supported with MCS buffer layout with these alignments in the RT space: Horizontal Alignment = 128 and Vertical Alignment = 64.</td></tr>
<tr><td>MCS and Lossless compression is supported for TiledY/TileYs/TileYf non-MSRTs only.</td></tr>
<tr><td>Clear is supported only on the full RT; i.e., no partial clear or overlapping clears.</td></tr>
<tr><td>The following table describes the RT alignment:
<table>
<tr><th>TiledY RT CL</th><th>Pixels</th><th>Lines</th></tr>
<tr><td>bpp</td><td></td><td></td></tr>
<tr><td>32</td><td>8</td><td>4</td></tr>
<tr><td>64</td><td>4</td><td>4</td></tr>
<tr><td>128</td><td>2</td><td>4</td></tr>
<tr><th>TiledX RT CL</th><th></th><th></th></tr>
<tr><td>bpp</td><td></td><td></td></tr>
<tr><td>32</td><td>16</td><td>2</td></tr>
<tr><td>64</td><td>8</td><td>2</td></tr>
<tr><td>128</td><td>4</td><td>2</td></tr>
</table>
</td></tr>
</table>

Restrictions
MCS buffer for non-MSRT is supported only for RT formats 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.
Clear rectangle must be aligned to two times the number of pixels in the table shown below due to 16X16 hashing across the slice.

TiledY RT	Pixels	Lines
bpp		
32	128	64
64	64	64
128	32	64

To optimize the performance MCS buffer (when bound to 1X RT) clear similarly to MCS buffer clear for MSRT case, clear rect is required to be scaled by the following factors in the horizontal and vertical directions:

MCS CL for TiledY RCC	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
32	64	32
64	32	32
128	16	32

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.

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.
- After Render target fast clear, pipe-control with color cache write-flush must be issued before sending any DRAW commands on that render target.

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.

A rectangle primitive must be scaled down by the following factors with respect to render target being resolved:

MSC CL for TiledY RCC	Horizontal scale down factor	Vertical scale down factor
bpp		
32	64	32
64	32	32
128	16	32

Programming Note	
Context:	Render Target Resolve

- **Depth Test Enable, Depth Buffer Write Enable, Stencil Test Enable, Stencil Buffer Write Enable,** and **Alpha Test Enable** must all be disabled.
- This render target resolve procedure is not supported on multisampled render targets. Unresolved multisampled render targets are directly supported by the sampling engine, which resolves clear values in addition to decompressing the surface This applies to both *ld2dms* and *sample2dms* messages.

Media GPGPU Pipeline

This section of the BSpec discusses Programming the GPGPU Pipeline, Thread Group Tracking, Generic Media, and other related topics.

Media GPGPU Pipeline]

This section of the BSpec discusses Programming the GPGPU Pipeline, Thread Group Tracking, Generic Media, and other related topics.

Programming the GPGPU Pipeline

1. In MEDIA_VFE_STATE choose whether to set DW2.6 Bypass Gateway Control. Usually this will be set, allowing the gateway to be used without OpenGateway/CloseGateway.
2. Set the virtual pools of EUs via MEDIA_POOL_STATE.
3. Set up interface descriptor with # of threads in barrier. The barrier id is not specified here because can Gen7 automatically assigns barriers to thread groups when they are free. The amount of CURBE data to deliver per thread dispatch is set in the interface descriptor.
4. Set up CURBE with thread ids and common data for all thread dispatches in the thread group.
5. Set up a GPGPU_WALKER command or a set of GPGPU_OBJECT commands with the thread group ids to dispatch the threads. The CURBE data is sent in sections for each thread dispatch in the thread group; a new thread group starts sending the CURBE data from the beginning of the buffer.

Note: Gen7 can either have the barriers and SLM automatically managed by hardware or specified by software. Mixing software managed and hardware managed in the same set of threads is allowed, but may cause stalls if there is an allocation conflict.

Note: When using GPGPU_OBJECT, finish dispatching a thread group before starting a different one.

The kernel should handle the barriers as follows:

The BarrierMsg message 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 N0.

If a GPGPU kernel is going to access the Render Cache then the Fixed Function DOP Clock Gate Disable should be set to prevent the clock gating of the DAP used to access the Render Cache. This should be reset as soon as all the kernels that are accessing the Render Cache are finished and the Render cache flushed.

GPGPU Thread Limits

GPGPU requires 1024 SIMD channels to be available for a maximum size thread group. In a HSW/GT2 system with 10 EUs per subslice, each with 7 hardware threads, this means that a maximum size thread group will fit in a subslice if SIMD16 instructions are used, but not if SIMD8 are used. These limits can be circumvented for thread groups which do not need access to a barrier or SLM, in which case the thread group can cross sub-slices.

The Configurations section should be referenced to determine what SIMD is required to fit in the subslice of the targeted configuration.

GPGPU Commands

This section contains various commands for GPGPU, including a number of them shared with media mode.

MEDIA_VFE_STATE with varying definitions for different generations and projects:

MEDIA_VFE_STATE

MEDIA_CURBE_LOAD

MEDIA_INTERFACE_DESCRIPTOR_LOAD

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address, with varying definitions for different generations and projects:

INTERFACE_DESCRIPTOR_DATA

MEDIA_STATE_FLUSH

GPGPU_WALKER

GPGPU Command Workarounds

GPGPU Commands have some subtle programming restrictions and workarounds to be aware of, as described below.

GPGPU Indirect Thread Dispatch

Indirect thread dispatch allows one thread group to control the group size of a following thread group.

This is the sequence of commands in the ring buffer:

GPGPU_OBJECT/WALKER	// Either a set of objects or a walker to dispatch a thread group which will 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 this data to memory:

1. The thread ids delivered in the CURBE - written where the following MEDIA_CURBE_LOAD will read them.
2. The GPGPU_WALKER parameters are written to memory where the MI_LOAD_REGISTER_MEMORY will read them.
    a. GPGPU_DISPATCHDIMX - the X dimension of the number of thread groups to dispatch in:

DWord
7

    b. GPGPU_DISPATCHDIMY - the Y dimension of the number of thread groups to dispatch in:

DWord
10

    c. GPGPU_DISPATCHDIMZ - the Z dimension of the number of thread groups to dispatch in:

DWord
12

See vol1c Memory Interface and Command Stream for the MMIO register addresses and formats.

If any of the indirect registers are set to 0 then no work will be dispatched.

GPGPU Context Switch

The GPGPU pipeline supports interruption of GPGPU workloads on thread group boundaries. This is needed for general purpose GPGPUs that are so large that there is a risk of the display becoming non-responsive if the work cannot be interrupted for other jobs.

A workload is interrupted with the MI_ARB_CHECK command with the UHPTR register. The MI_ARB_CHECK command is placed throughout the command buffer. The driver updates the UHPTR register when a new context is needed; MI_ARB_CHECK checks for this and reprograms the head and tail pointers to the new batch of commands. The driver waits for the preemption to occur without going into RS2.

The GPGPU needs to modify this to allow a GPGPU_WALKER command to be interrupted. This is done by following each GPGPU_WALKER command with a MEDIA_STATE_FLUSH. This causes the CS to stop fetching commands until either the command completes or until the UHPTR valid bit is set.

GPGPU workloads can be dispatched with either GPGPU_OBJECT commands or GPGPU_WALKER commands. In the case of GPGPU_OJBECT, the MEDIA_STATE_FLUSH/ MI_ARB_CHECK pair must be placed in the batch buffer at thread group boundaries, since preemption cannot occur with a thread group partially dispatched. GPGPU_WALKER commands can dispatch multiple thread groups, in this case the MEDIA_STATE_FLUSH/ MI_ARB_CHECK follows each GPGPU_WALKER and the hardware takes care of noticing the UHPTR update and stopping at the next thread group boundary.

The commands in the batch buffer will look something like this:

Command Ring	Notes
MI_SET_CONTEXT	Go to GPGPU context
MI_BATCH_BUFFER_START	If new context, set address to top of batch. Otherwise, address needs to be set to the command preempted (given in the HWSP). The GP GPGPU bit must be set.

Command Batch	Notes
GPGPU_OBJECT	
GPGPU_OBJECT	
...	(more threads forming a complete thread group)
MEDIA_STATE_FLUSH	Check for preemption at thread group boundary. “Preemption” defined by the UHPTR valid bit set.
MI_ARB_CHECK	Move the head only if UHPTR valid bit is set.
...	
GPGPU_WALKER	
MEDIA_STATE_FLUSH	Check for preemption at thread group boundary internal to GPGPU_WALKER command. “Preemption” defined by the UHPTR valid bit set.
MI_ARB_CHECK	Move the head only if UHPTR valid bit is set.
...	

Command Batch	Notes
MI_BATCH_BUFFER_END	GPCS batch workload bit is cleared.

The context saved will consist of the state commands for VFE and a modified GPGPU_WALKER command with a new starting thread group id. On context restore, the commands are executed to start the GPGPU_WALKER where it left off before continuing with the rest of the command buffer.

An example software model for starting a preemption goes like this:

1. The UHPTR is reprogrammed to point to the current tail of the ring buffer.
2. Insert new commands:
    a. LRI to UHPTR to clear valid.
    b. Store Register to mem the preempted batch offset.
    c. Store Register to mem the preempted ring offset.
    d. Pipe_control notification.
    e. An MI_SET_CONTEXT to the new context is put into the ring.
3. Insert commands for new context. i.e. batch buffers.
4. Update Tail Pointer.

Programming Note	
Context:	GPGPU Context Switch
• 2-3 items above could happen during execution of a thread group so the HW may see the tail pointer updated before preemption starts. • The driver needs to turn off RC6 during items 1 and 4.	

GPGPU Context Switch

Context switch for BDW onwards allows the switch to take place in the middle of a thread group to provide better response time.

The command sequence for BDW has been simplified – MEDIA_STATE_FLUSH and MI_ARB_CHECK are now optional between commands for preemption to occur. MEDIA_STATE_FLUSH is now only needed before MEDIA_LOAD_CURBE commands to ensure CURBE is done being read before reloading it. The watermark bit in MEDIA_STATE_FLUSH is not needed, since the check is done automatically before a thread group is started.

Preemption can occur on commands listed here:

- MI_ARB_CHECK
- MEDIA_STATE_FLUSH
- PIPE_CONTROL
- MI_WAIT_FOR_EVENT
- MI_SEMAPHORE_WAIT

- GPGPU_WALKER – Preemption can occur at any time, with thread groups partially complete; the system state is saved/restored for context save and restore
- MI_WAIT_FOR_EVENT
- MI_SEAMPHORE_WAIT

Messages to the Sampler must use headers (controlled by bit 19 of the Message Descriptor) when pre-emption is enabled.

See GPGPU Context Switch Workarounds for information about programming restrictions and workarounds.

Note that command preemption is not supported for MEDIA_OBJECT_* commands for MI_ARB_ON/OFF should be used to prevent preemption except at frame boundaries, where an MI_ARB_CHECK should be inserted.

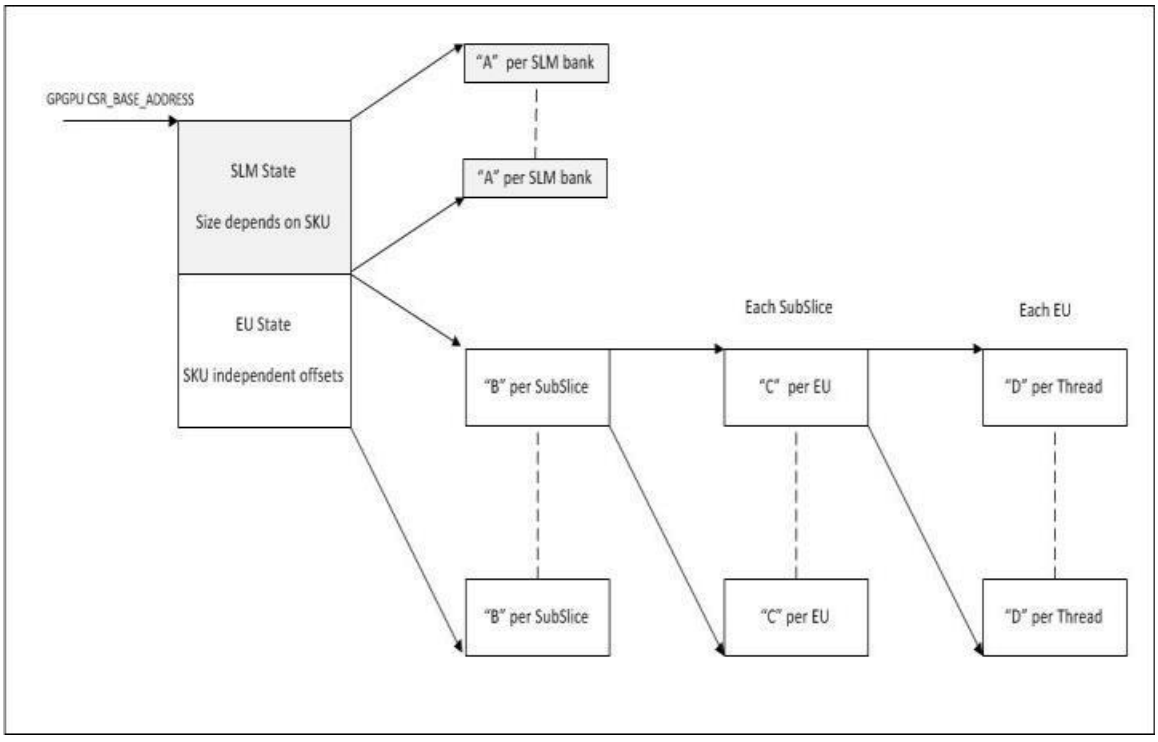
The memory map of the context image that is saved from GPGPU pipeline includes SLM and EU State. A contiguous space is allocated to save the SLM and EU State starting from the address provided in GPGPU_CSR_BASE_ADDRESS.

Maximum Upper Bound

In GPGPU context save/restore mode, hardware writes to this location and does NOT use the surface state or scratch space.

Memory Map of the GPGPU Context Image

The address offsets computed by hardware depend on the number of slices, subslices, EUs, and threads.



Base Address Calculation

Base Address = CSR_Base_Address + A* [Config.NumSlmBanks] + B*SubSliceId + C*EuId + D*ThreadId + Message_Offset

A = 0x10000 // 64K SLM per SubSlice

B = C * [Config.NumEusPerSubSlice]

C = D * [Config.NumThreadsPerEu]

D = 0x2000 // 8KB

Note: The slicenumber and EUIDs may require re-mapping such that a contiguous space is used with no gaps inbetween.

GPGPU Context Switch Workarounds

The GPGPU Context Switch support has some subtle programming restrictions and workarounds, as described below.

Media GPGPU Payload Limitations

There are 3 types of payload that the media/GPGPU instructions can have, but not all of them are allowed. The following table lists the legal combinations:

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 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

Indirect and CURBE payloads are fetched during thread dispatch from memory using the Dynamic State MOCS (specified in [Instruction] STATE_BASE_ADDRESS).

Synchronization of the Media/GPGPU Pipeline

The Media/GPGPU Pipeline is synchronized in the same way as the 3D pipeline using the PIPE_CONTROL command.

See the Bspec section on 3D pipe synchronization: vol2a 3D Pipeline - Overview > 3D Pipeline > Synchronization of the 3D Pipeline.

Mode of Operations

This section describes the dispatch and payload of the GPGPU threads.

GPGPU Mode

The general purpose (GPGPU) mode allows the Gen7 architecture to be used by general purpose parallel APIs:

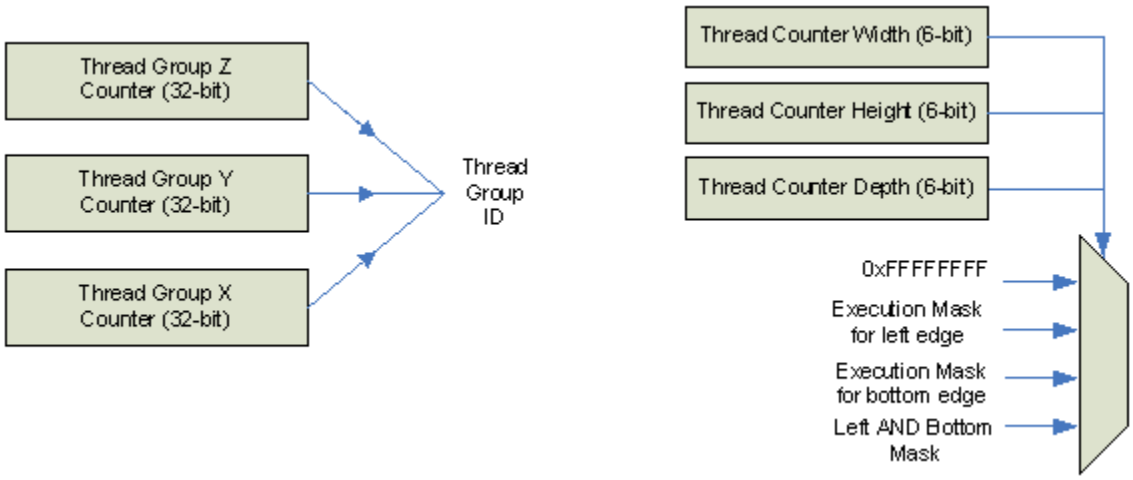
- OpenCL
- DX11 ComputeShader

This is similar to the Generic mode with additional support for automatic generation of threads, Shared Local Memory, and Barriers.

Automatic Thread Generation

A single GPGPU job may require thousands or even millions of GPU_OBJECT commands. Rather than create them separately, it would be better to generate them algorithmically. To do this a GPGPU_WALKER command is created.

Rather than modifying the Media Walker, a simple Thread Group Walker is created instead:



The X/Y/Z counters for the thread group will have an initial and maximum value. The thread group ID sent with each dispatch consists of these 3 numbers. These counters are 32 bits since the spec does not limit the size of the thread ID.

The 3 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 3 thread counters in order to select the execution masks correctly – see the section *Execution Masks* . Each one is 6 bits to allow full flexibility of any dimension going to 64 while the rest do not increment.

A thread is generated each time one of the thread counters increment. When all the counters reach their maximum values, the thread group is done and the thread group counter can increment and start a new thread group. When the thread group X counter reaches its maximum it is reset to 0, and the Y counter is incremented.

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.

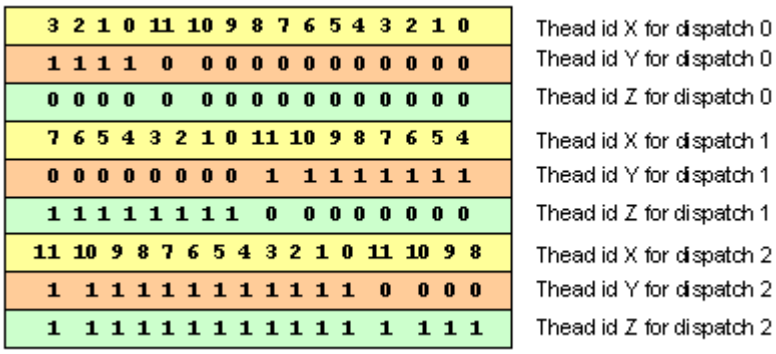
Thread Payload

The payload to each thread dispatched is:

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

Thread IDs form a 2D or 3D surface which has to be mapped into SIMD32, SIMD16 or SIMD8 dispatches. Rather than have the hardware force a particular mapping of thread IDs to channels, the mapping will be supplied by the compiler. The VFE will receive a simple count of the number of threads per thread group which will be used to count the number of dispatches. The thread IDs for all threads in a thread group are put in a constant buffer with the MEDIA_CURBE_LOAD command. A single set of thread IDs can be used repeatedly for all thread groups, since the thread IDs are the same for each thread group ID output by the GPGPU_WALKER.

The data required is up to the compiler, but here is an example set of payloads for a 2 Z x 2Y x 12 X and a SIMD16 dispatch. This thread group requires 3 dispatches:



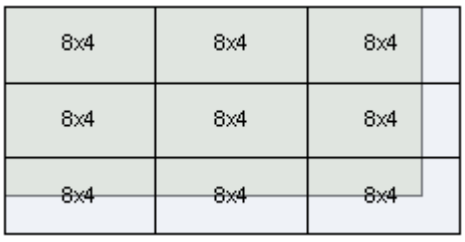
In this case the thread counter width would be programmed with a maximum value of 3 (since all the execution masks are all F, it doesn't matter how the thread counters are programmed as long as they count to 3 before finishing the thread group).

The first dispatch would tell the TS (who would tell the TD) that the payload starts at the beginning of the constant buffer and has a length of 3. 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 get exactly the same payload – the only difference is the thread group ID (as well as a different barrier and shared local memory space).

Execution Masks

The number of channels required by the GPGPU job may not evenly fit into the number of SIMD channels. That can leave some channels idle. The execution mask is used to tell the hardware which channels are to be used.

A thread group is modeled as a 3D solid with each channel acting as one X/Y/Z point in the solid. This can take the form of a line with 1024 channels with X from 0 to 1023 and constant Y/Z, a square with X=0 to 32 and Y=0 to 32, or a cube with X=0 to 9, Y=0 to 9, Z=0 to 9. Software needs to determine how these shapes are mapped onto the 32 SIMD32 channels per dispatch (or 16 SIM16, etc). The mapping per thread is assumed to be a 2D square of channels such as 8x4, 16x2, 32x1. Below is a diagram of a 22x6 thread group that is mapped onto a set of 8x4 SIMD32 channels:



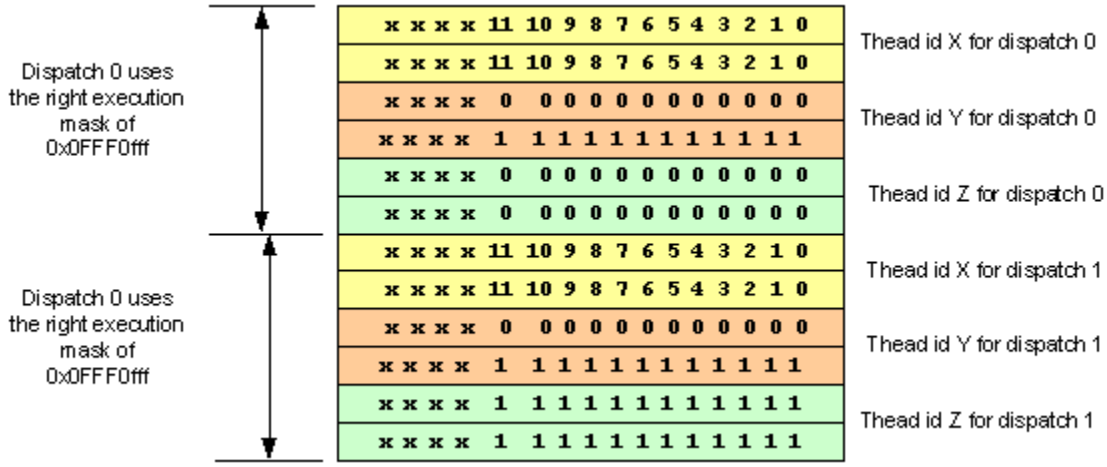
Note that the dispatches to the top and left have execution masks of all-F, while all the right edge dispatches have the same execution mask; likewise all the bottom edge dispatches have the same execution mask. The bottom right is the logical-AND of the right and bottom edge dispatches.

A 32-bit right and bottom mask is sent with the GPGPU_WALKER command, and the thread width, height and depth counters are used to determine when they 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 is incremented until it reaches the maximum – the dispatch with width=max will use the right execution mask. The height counter is then incremented and 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 will be the AND of the right and bottom and the depth counter will increment.

The same 2Z x 2Y x 12X thread group described above dispatched as SIMD32 with each dispatch delivering a 16X x 2Y shape would require 2 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 1, and the depth counter would have a maximum of 2. The two dispatches would 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:

In this case the width counter can have a maximum of 2, and the height and depth counters with a maximum of 1. The first dispatch will use the bottom mask only (all-F) and the second will use the right AND bottom mask to remove the channels that are not used.

URB Management

The VFE manages the URB in GPGPU and generic/media modes.

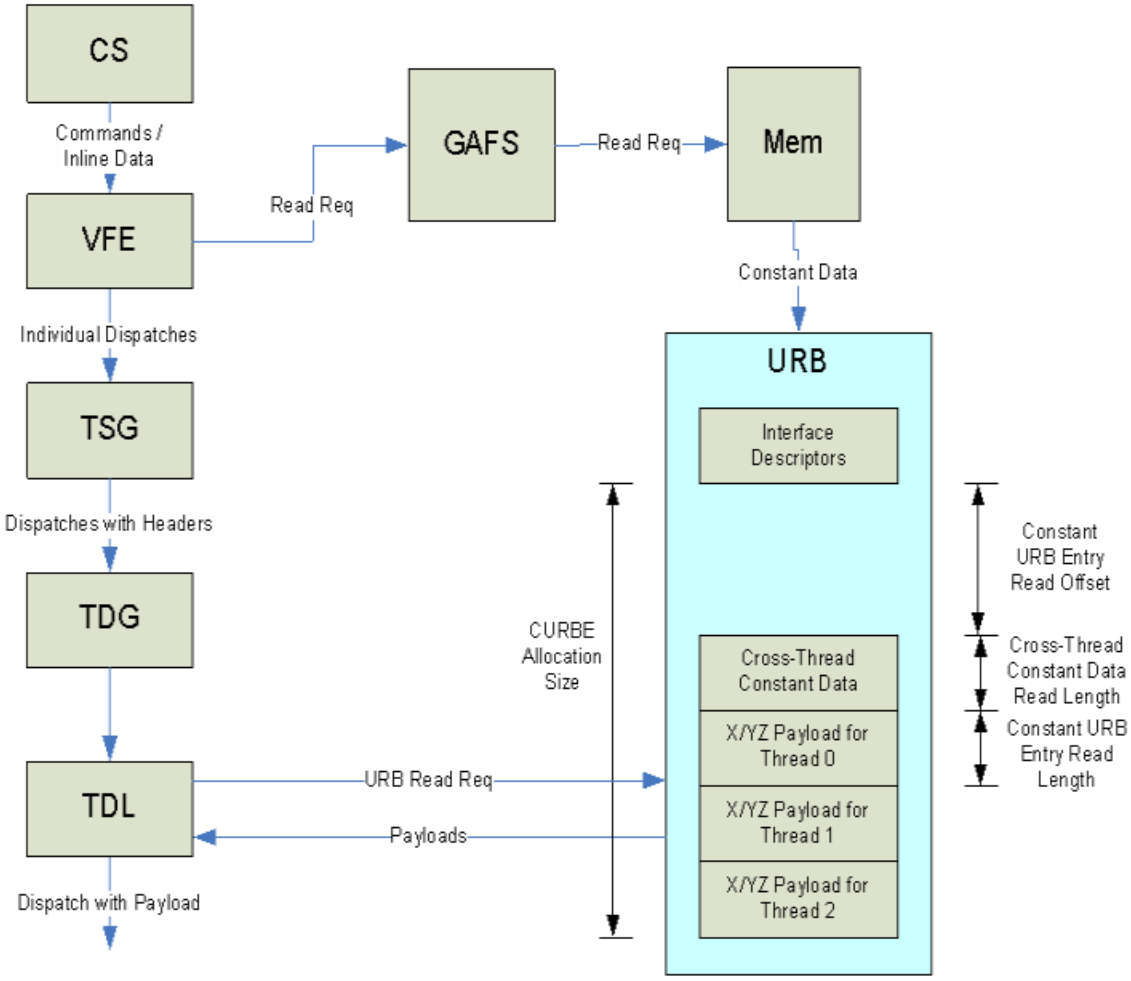
Description
The first 64 URB entries are reserved for the interface description, and CURBE data is placed after the IDs. URB handles are needed for indirect data and parent/child communication; when the VFE starts up it creates up to 128 handles by partitioning the remaining URB space into evenly spaced addresses and saving the resulting handles in a FIFO. The handles can then be treated just like ones created by the URBM - send to TD on dispatch and recovered on the handle return bus. MEDIA_VFE_STATE specifies the amount of CURBE space, the URB handle size and the number of URB handles. The driver must ensure that: ((URB_handle_size * URB_num_handle) – CURBE – 64) <= URB_allocation_in_L3.

Indirect Payload Storage

The GPGPU commands are extended to allow indirect input as an alternative to CURBE. The mechanism used for CURBE will be used in exactly the same way for indirect: the same offset is used which specifies what data is delivered to all threads, and then a count which specifies how much data is delivered per thread. A single indirect pointer points to both the Cross-Thread and Per-Thread Constant Data, which is stored in the URB. The position of the Cross-Thread and Per-Thread constants are swapped in the EU GRF.

To use indirect payload storage, the (URB Entry Allocation Size * Number of URB Entries) product must be enough to cover the sum of the Cross-Thread and Per-Thread Indirect data to be loaded. These state variables are set in the MEDIA_VFE_STATE command. The URB Entry Allocation must be equal or greater than th GPGPU_WALKER's Indirect Data Length. In general the maximum number of URB Entries that fills the URB space should be used to ensure that a shortage of handles doesn’t cause a performance problem.

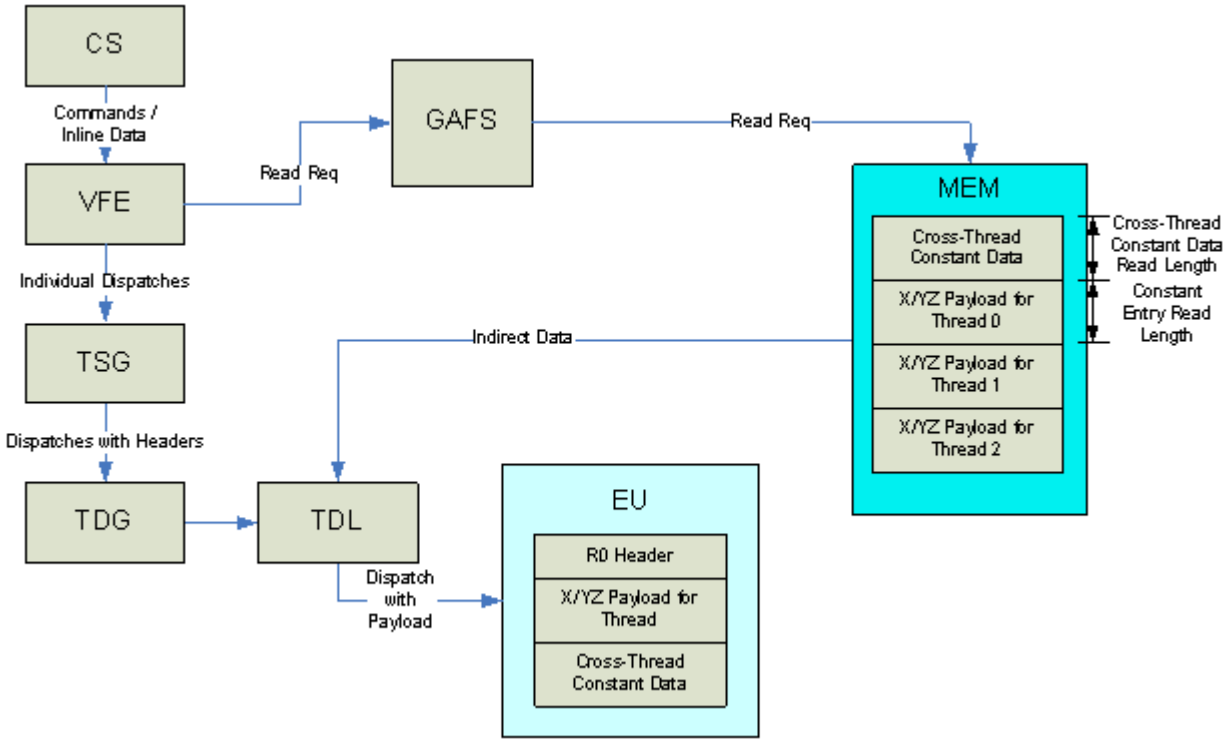
CURBE Payload



Example of CURBE command sequence:

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

Figure for Indirect Payload



Example of Indirect Command Sequence:

```
GPGPU_WALKER // Indirect pointer included in command
GPGPU_WALKER // No need to flush between commands for CURBE.
```

The differences between CURBE and Indirect are:

- The indirect uses a 32-bit memory pointer for the start address.
- The Constant URB Entry Read Offset is not used.
- The Cross-Thread Constant Data Read Length and the Constant Entry Read Length is multiplied by 32 to convert it into bytes.
- The X/Y/Z payload in the EU GRF comes before the Cross-Thread Constant Data.

MEDIA_OBJECT_GRPID

GPGPU_OBJECT already allows barriers and SLM to be allocated, but does not allow the scoreboard to be used. We need a single command which allows both.

A new command called MEDIA_OBJECT_GRPID is created which is the same as the basic MEDIA_OBJECT command with the addition of a 32-bit group id and an end-of-group bit. The group id will be used to allocate barriers and SLM based on the Interface Descriptor.

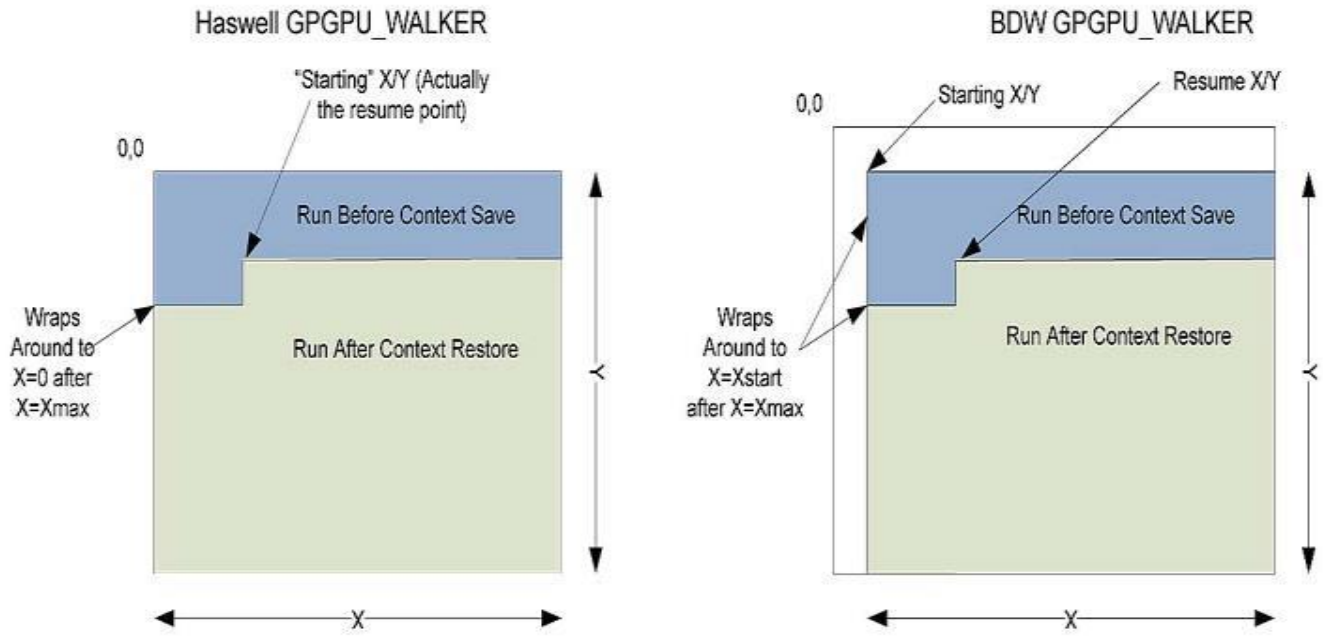
Restrictions that are currently in place for GPGPU would carry over to Media use of these features:

- All threads that use them must be on the same Tslice unless cross-slice barriers are enabled.

- We only dispatch a single group at a time – this is naturally supported by using only the most outer loop numbers for the in MEDIA_OBJECT_WALKER, but must be enforced by the programmer for MEDIA_OBJECT_GRPID.

Starting Offset for a Thread Group ID

The GPGPU_WALKER command has starting values for the Thread Group IDs that the walker generates per thread group, but they are designed to allow context save and restore rather than provide a true initial value. When walking a 2D set of groups in X/Y, when the X value reaches the maximum, the Y increments and the next X is 0, rather than going back to Xstart. This behavior is expected on context restore where the true initial values are 0,0 and the starting values are actually the resume values of the restore.



GPGPU Thread R0 Header

The RO header of the Thread Dispatch Payload for the GPGPU thread:

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]
	9	**GPGPU Dispatch.** Indicates that this dispatch is from the GPGPU pipe (see PIPELINE_SELECT command).
	8:0	**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. 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.

DWord	Bits	Description
R0.2	31	**Barrier ID MSB:** This field is bit[4] of the BarrierID; for full 5-bit barrier ids it should be combined with Barrier ID[3:0]. Format: U1
	30	Reserved: MBZ
	29	**Barrier Enable:** This field indicates that a barrier has been allocated for this kernel.
	28	**SLM Enable:** This field indicates that Shared Local Memory has been allocated for this kernel.
	27:24	**BarrierID:** This field indicates the barrier that this kernel is associated with. Format: U4
	14:11	Reserved: MBZ
	10	Reserved: MBZ
	3:0	Reserved: MBZ
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	31:30	Reserved: MBZ
	29:24	**Shared Local Memory Index:** Indicates the starting index for the shared local memory for the thread group. Format = U6
	23:16	Reserved: MBZ
	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.

Thread Group Tracking

The TSG needs to keep track of the threads outstanding in a group to know when the thread group barrier and Shared Local Memory can be reclaimed. This can be done by keeping a counter per active thread group (up to 16 per half-slice) which increments when a new thread is sent out and decremented when the thread retires. The assigned barrier ID (with half-slice bit) is unique per thread group and much smaller than the thread group ID and so will be used to keep track of the thread group instead.

Since TSL sends the thread retirement via the Message Channel rather than the thread retirement bus, the barrier ID used to identify the thread group can be sent at the same time. A CAM will then match the ID with the counter to decrement.

There is a potential corner case of a thread group without barriers being partly dispatched, then retiring before the rest of the thread group is sent. This should be OK, since the lack of barriers means that there are no dependencies between threads.

Shared Local Memory Allocation

The Shared Local Memory is a 64k block per half-slice in the L3 that must be shared between all thread groups on that half-slice. A new memory manager simular to the Scratch Space memory manager is used to allocate this space.

We are only dispatching threads from a single Interface Descriptor at a time. If a new Interface Descriptor is requested the pipe is drained and all shared memory recovered before starting to allocate new shared memory. This means that only a single size of shared memory needs to be supported at once.

For simplicity, only power-of-2 sizes from 4k to 64k are allowed. The thread request will specify how much is needed. The first thread of a Thread Group is marked as requiring a new shared local memory – if not the old Shared Local Memory offset is sent with the dispatch.

A simple set of 16-bits is used to allocate 4k shared memory, with fewer bits used for larger sizes. A priority encoder finds the first unused bit and the offset remembered as being associated with a particular barrier id. The barrier id is then used to track the thread group.

When the Thread Group Tracking indicates that a thread group is completely retired, that section of shared local memory can be reclaimed.

Software Managed Shared Local Memory

Software can optionally manage shared local memory. In this case, each thread command or thread group command will have the shared memory offset included – each command in a thread group must have the same offset, of couse. If the offset requested is still being used then the command is stalled until the thread group using that offset is done.

Hardware will track the usage of this section of shared memory as before, recording the offset as being used and recording it as being available after the thread group is done.

Automatic Barrier Management

Description
Since we have an automatic shared memory allocation it makes sense to make barrier management automatic too. Instead of the barrier id in the Interface Descriptor, there is now a thread count per thread group.
If a new thread group id comes in without a barrier allocated (checked with a CAM match across 32 barriers per subslice), the TSG picks a unused barrier and sends this count in a message to GWunit. It then needs to wait for an accept message back from GW before sending the dispatch to ensure that a barrier message doesn't arrive at the GW before the barrier is programmed. The barrier ID picked is sent with every dispatch from this thread group.
When the thread group tracker determines that a thread group has finished, the barrier becomes available to new thread groups.

Dispatch Payload

The payload for a general purpose thread will have to include the execution mask with a bit per 32-channel. SIMD16 and SIMD8 use the LSB bits of the execution mask. The 5-bit number transferred from VFE will be expanded to produce the 32-bit mask. This will use the Dmask currently used by the pixel shader dispatch in the transparent header.

Cross-Slice Barriers

Barriers in earlier projects were limited to communicating with EUs only in the local subslice. This works for GPGPU workloads, but Media workloads don't always fit in a subslice, and would be much improved with the ability to use barriers to synchronize across threads.

When a barrier is enabled with the global barrier bit set and SLM is disabled, the MEDIA_OBJECT_WALKER and MEDIA_OBJECT_GRPID can allow groups to cross slices. If SLM is enabled, then the restriction described in the following table applies:

Restriction
If SLM is enabled, then the group must stay on a single subslice or pool.

Each barrier will be marked as either local or global. If global it will force all subslices to use the same barrier id to lower gatecount in TSG – though this might cause a delay if a bunch of local barriers are allocated and no single global id is available across the needed subslices.

If the barrier is marked as global, the TSG will know that that group does not need to be forced into a single subslice. It will know from the interface descriptor how many threads are part of the barrier, but it will have to split this number among the subslices. TSG will dispatch as many threads as possible to a single subslice before going to the next to keep the number of subslices involved as low as possible. TSG will need to wait until there is enough threads available in the global barrier for the entire group before starting the dispatch so that it will know beforehand what to program the local subslice barrier to. Note that reprogramming the number of threads in a barrier in GW with the EU through the barrier message will not work in this case, since the EU can only reprogram the local GW.

Each subslice is programmed with the local number of threads and will work independently - when the local set of threads has been met, the GW will send a message to TSG. TSG has a simplified version of the

barrier logic - it has a count of how many GWs it expects to report per barrier (up to 8) and a count of how many have reported so far. It can use a simple bit-mask to record the GWs that have reported back so far. When all the gateways have reported, it will send a message back to each saying telling them to complete the barrier.

This scheme is slower than a normal barrier, but should be much faster than using atomics and polling to synchronize threads.

Programming Note	
Context:	Cross-Slice Barriers
When using cross-slice barriers, enough subslices must be available to run the group using the barrier - slice shutdown for power must be limited to ensure this.	

Generic Media

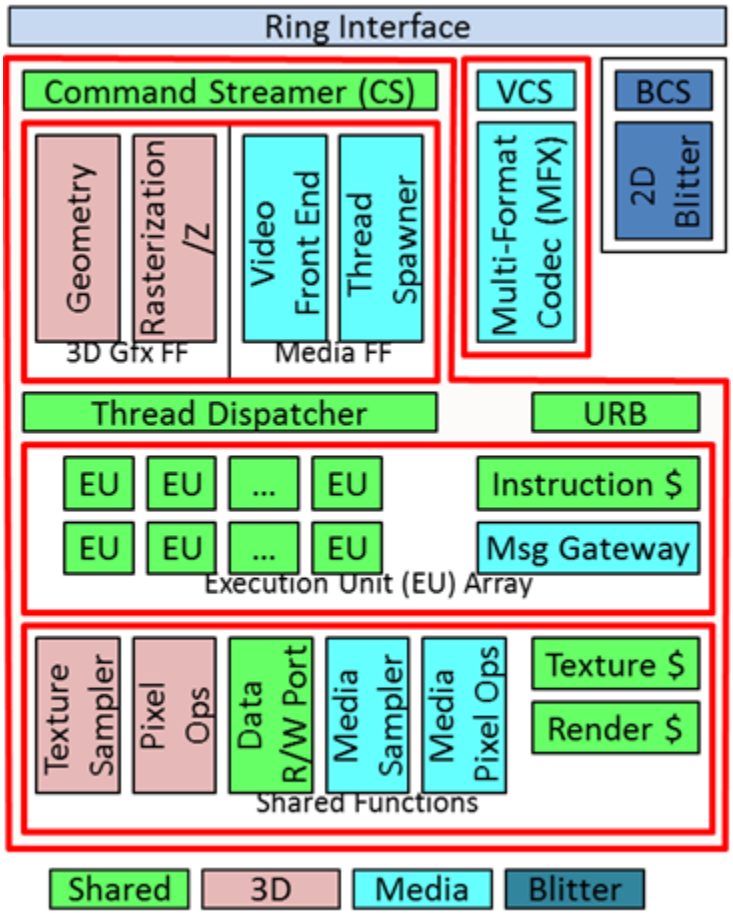
This introduction provides a brief overview of the Media product features. It includes Media functions, feature benefits, and how the features fit into graphics products as part of a whole solution.

Media product features include:

- Multi-format codec engine
- Video front end
- Media fixed functions
- Video encoding
- Video decoding
- Sampling

These product features support specific applications, such as interactive gaming, videogames, social media, virtual reality, and augmented reality.

The following block diagram shows the Main Render Engine, unified for 3D graphics and Media.



- **Fixed Function (FF) pipelines:** Provide thread generation and control.
- **3D graphics or Media FF:** Controls EU array at a given time. The EU (Execution Unit) array is shared between 3D and Media; ISA is optimized for both.
- **Shared functions:** Are accelerators to run filtered load, scatter, gather, and filter/blended store operations.
- **MFX:** Is a parallel codec engine that runs in a separate context.

Media and General Purpose Pipeline

Introduction

This section covers the programming details for the media (general purpose) fixed function pipeline. The *media pipeline* is positioned in parallel with the 3D fixed function pipeline. It provides media functions and has media specific fixed function capability. However, the fixed functions are designed to have the general capability of controlling the shared functions and resources, feeding generic threads to the Execution Units to be executed, and interacting with these generic threads during run time. The media pipeline can be used for non-media applications, and therefore, can also be referred to as the *general purpose pipeline*. For the rest of this chapter, we refer to this fixed function pipeline as the media pipeline, keeping in mind its general purpose capability.

Concurrency of the media pipeline and the 3D pipeline is not supported. In other words, only one pipeline can be activated at a given time. Switching between the two pipelines within a single context is supported using the MI_PIPELINE_SELECT command.

Following are some media application examples that can be mapped onto the media pipeline. All these applications are functional; however, the level of performance that can be achieved depends on the hardware configuration and is beyond the scope of this document.

Application
MPEG-2 decode acceleration with HWMC (e.g. DXVA HWMC interface)
MPEG-2 decode acceleration with IS/IDCT and forward (e.g. DXVA IDCT interface)
MPEG-2 decode acceleration with VLD and forward (e.g. DXVA VLD interface)
AVC decode acceleration with HWMC and forward including Loop Filter
VC1 decode acceleration with HWMC and forward including Loop Filter
Advanced deinterlace filter (motion detected or motion compensated deinterlace filter)
Video encode acceleration (with various level of hardware assistance)

Terminology

Term	Description
AVC	Advanced Video Coding. An international video coding standard jointly developed by MPEG and ITU. It is also known as H.264 (ITU), or MPEG-4 Part 10 (MPEG).
Child Thread	A thread corresponding to a leaf-node or a branch-node in a thread generation hierarchy. All thread originated from kernels running on the execution units are child threads.
EOB	End of Block. It is a 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer.
IDCT	Inverse Discrete Cosine Transform. It is the stage in the video decoding pipe between IQ and MC.
ILDB	In-loop Deblocking Filter – the deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe. It is required to support VC-1 Profile B video decoder acceleration.

Term	Description
IQ	Inverse Quantization. It is a stage in the video decoding pipe between IS and IDCT.
IS	Inverse Scan. It is a stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for MPEG-2 and VC-1.
IT	Inverse Integer Transform. It is the stage in AVC or VC-1 video decoding pipe between IQ and MC.
MPEG	Motion Picture Expert Group. MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
MC	Motion Compensation. It is part of the video decoding pipe.
MVFS	Motion Vector Field Selection – a four-bit field selecting reference fields for the motion vectors of the current macroblock.
PRT	A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread. Only one PRT is allowed in the system. Hardware is responsible for re-dispatching the incomplete PRT at context restore, and a PRT can continue operations from that previously left-over state.
Parent Thread	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Root Thread	A thread corresponding to a root-node in a thread generation hierarchy. In the general-purpose pipeline, all threads originated from VFE unit are root threads.
Synchronized Root Thread	A root thread that is dispatched by TS upon a 'dispatch root thread' message.
TS	Thread Spawner. It is the second (and the last) fixed function in the general-purpose pipeline.
Unsynchronized Root Thread	A root thread that is automatically dispatched by TS.
VC-1	VC-1 is the common name for the open video compression standard SMPTE 421M, adopted on April 3, 2006.
VFE	Video Front End. It is the first fixed function in the general-purpose pipeline.
VLD	Variable Length Decode. It is the first stage of the video decoding pipe that consists mainly of bit-wide operations. Hardware MPEG-2 VLD acceleration is supported in the VFE fixed function stage.

Hardware Feature Map in Products

The following table lists the hardware features in the media pipe.

Video Front End Features in Device Hardware

Features/ Device	
Generic Mode	**Y**
Root Threads	**Y**
Parent/Child Threads	**Y**
SRT (Synchronized Root Threads)	**Y**
PRT (Persistent Root Thread)	**Y**
Interface Descriptor Remapping	N
IS Mode (HW Inverse Scan)	N
VLD Mode (HW MPEG2 VLD)	N
AVC MC Mode	N
AVC IT Mode (HW AVC IT)	N
AVC ILDB Filter (in Data Port)	N
VC1 MC Mode	N
VC1 IT Mode (HW VC1 IT)	N
Stalling HW Scoreboard	**Y**
Non-stalling HW Scoreboard	**Y**
HW Walker	**Y**
HW Timer	**Y**
Pipelined State Flush	**Y**
HW Barrier	**Y**

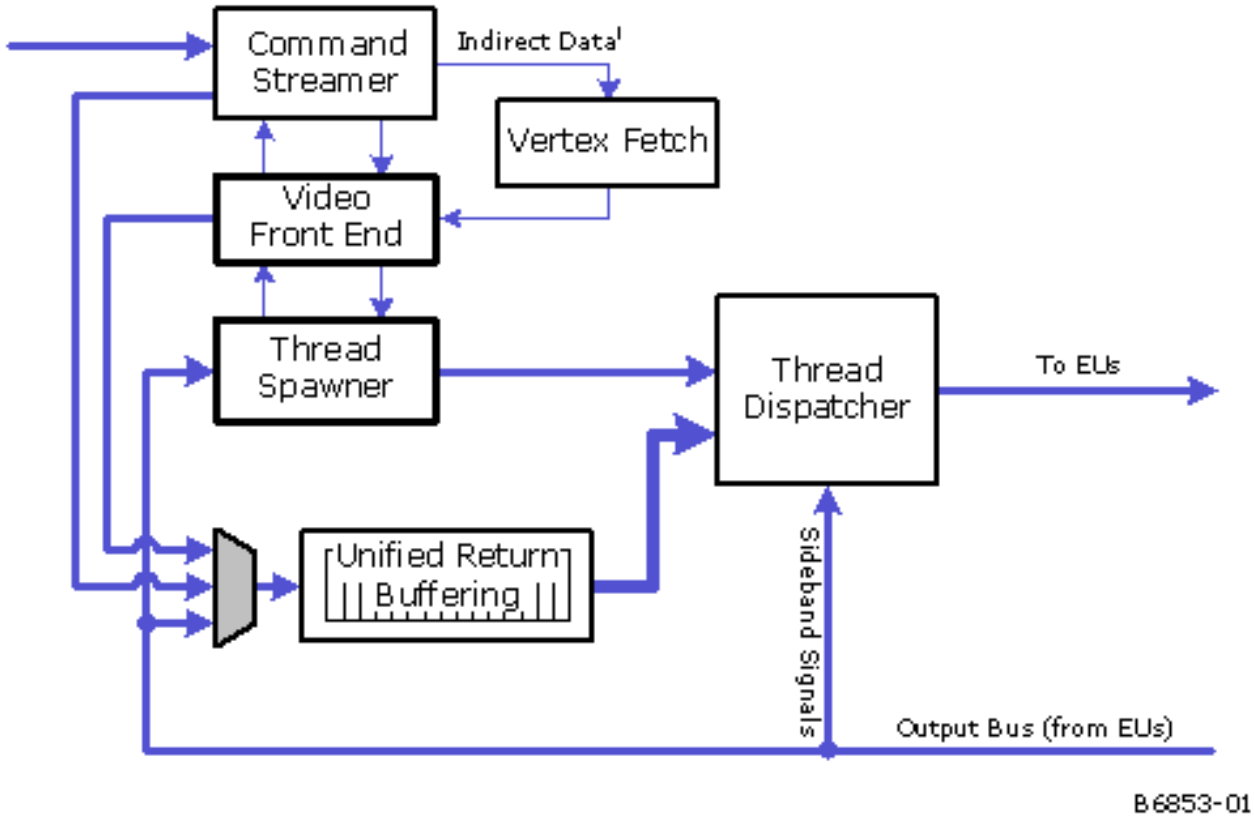
Media Pipeline Overview

The media (general purpose) pipeline consists of two fixed function units: Video Front End (VFE) unit and Thread Spawner (TS) unit. VFE unit interfaces with the Command Streamer (CS), writes thread payload data into the Unified Return Buffer (URB), and prepares threads to be dispatched through TS unit. VFE unit also contains a hardware Variable Length Decode (VLD) engine for MPEG-2 video decode. TS unit is the only unit of the media pipeline that interfaces to the Thread Dispatcher (TD) unit for new thread generation. It is responsible for spawning root threads (short for the root-node parent threads) originated from VFE unit and for spawning child threads (can be either a leaf-node child thread or a branch-node parent thread) originated from the Execution Units (EU) by a parent thread (can be a root-node or a branch-node parent thread).

The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- VFE manages URB and only has write access to URB; TS does not interface to URB.
- When URB Constant Buffer is enabled, VFE forwards TS the URB Handler for the URB Constant Buffer received from CS.
- TS interfaces to TD; VFE does not.
- TS can have a message directed to it like other shared functions (and thus TS has a shared function ID), and it does not snoop the Output Bus as some other fixed functions in the 3D pipeline do.
- A root thread generated by the media pipeline can only have up to one URB return handle.
- If a root thread has a URB return handle, VFE creates the URB handle for the payload to initiating the root thread and also passes it alone to the root thread as the return handle. The root thread then uses the same URB handle for child thread generation.
- If URB Constant Buffer is enabled and an interface descriptor indicates that it is also used for the kernel, TS requests TD to load constant data directly to the thread's register space. For root thread, constant data are loaded after R0 and before the data from the other URB handle. For child thread, as the R0 header is provided by the parent thread, Thread Spawner splits the URB handles from the parent thread into two and inserts the constant data after the R0 header.
- A root thread must terminate with a message to TS. A child thread should also terminate with a message to TS.
- High streaming performance of indirect media object load is achieved by utilizing the large vertex cache available in the Vertex Fetch unit (of the 3D pipeline).

Top level block diagram of the Media Pipeline



Generic Mode

In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. As there is no special fixed function logic used, the Generic mode can also be viewed as a 'pass-through' mode. In this mode, VFE generates a new thread for each MEDIA_OBJECT command. The payload contained in the MEDIA_OBJECT command (inline and/or indirect) is streamed into URB. The interface descriptor pointer is computed by VFE based on the interface descriptor offset value and the interface descriptor base pointer stored in the VFE state. VFE then forwards the interface descriptor pointer and the URB handle to TS to generate a new root thread. Many media processing applications can be supported using the Generic mode: MPEG-2 HWMC, frame rate conversion, advanced deinterface filter, to name a few.

Programming Media Pipeline

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

Command Sequence

Media pipeline uses a simple programming model. Unlike the 3D pipeline, it does not support pipelined state changes. Any state change requires an MI_FLUSH or PIPE_CONTROL command. When programming the media pipeline, it should be cautious to not use the pipelining capability of the commands described in the Graphics Processing Engine chapter.

To emphasize the non-pipeline nature of the media pipeline programming model, the programmer should note that if any one command is issued in the "Primitive Command" step, none of the state commands described in the previous steps cannot be issued without preceding with a MI_FLUSH or PIPE_CONTROL command.

Note for : With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. The MEDIA_STATE_FLUSH serves as a fence for state change by flushing the VFE/TS front ends but not waiting for threads to retire.

The basic steps in programming the media pipeline are listed below. Some of the steps are optional; however, the order must be followed strictly. Some usage restrictions are highlighted for illustration purpose. For details, refer to the respective chapters for these commands.

Command Sequence

For , the media pipeline is further simplified with fixed functions like MPEG2 VLD and AVC/VC1 IT removed. The addition includes:

1. The CURBE command is now unique to the media pipeline.
2. The interface descriptors are delivered directly as a media state command instead of being loaded through indirect state.

The programming model requires the following steps:

Step 1: MI_FLUSH/PIPE_CONTROL:

- This step is mandatory.
- Multiple such commands in step 1 are allowed, but not recommended for performance reasons.

Step 2: State command PIPELINE_ SELECT:

- This step is optional. This command can be omitted if it is known that within the same context the media pipeline was selected before Step 1.
- Multiple such commands in step 2 are allowed, but not recommended for performance reasons.

Step 3: State commands configuring pipeline states:

- STATE_BASE_ADDRESS:
  - This command is mandatory for this step (i.e. at least one).
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
  - This command must precede any other state commands below.
  - Particularly, the fields **Indirect Object Base Address** and **Indirect Object Access Upper Bound** are used to control indirect Media object load in VF.
  - The fields **Dynamics Base Address** and **Dynamics Base Access Upper Bound** are used to control indirect Curbe and Interface Descriptor object load in VF.
  - **Note:** This command may be inserted before (and after) any commands listed in the previous steps (Step 1 and 2). For example, this command may be placed in the ring buffer while the others are put in a batch buffer.
- STATE_SIP:
  - This command is optional for this step. It is only required when SIP is used by the kernels.
- MEDIA_VFE_STATE:
  - This command is mandatory for this step (i.e. at least one).
  - This command cause destruction of all outstanding URB handles in the system. A new set of URB handles will be generated based on state parameters, no. of URB and URB length, programmed in VFE FF state.
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- MEDIA_CURBE_LOAD:
  - This command is optional.
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- MEDIA_POOL_STATE:
  - This command is advisable as it sets the virtual pool sizes of EUs to contain workgroup execution.
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- MEDIA_INTERFACE_DESCRIPTOR_LOAD:
  - This command is mandatory for this step (i.e. at least one).
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.

Step 4: Primitive commands:

- MEDIA_OBJECT:
  - This step is optional, but it does not make practical sense to not issue media primitive commands after going through the previous steps to set up the media pipeline.
  - Multiple such commands in step 4 can be issued to continue processing media primitives.

With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. To support context switch for barrier groups, watermark and barrier dependencies are added to the MEDIA_STATE_FLUSH command. The usage of barrier group may have strict restriction that all threads belonging to a barrier group must all be present to avoid deadlock during context switch. Here are the example programming sequences to allow context switch.

Notes
The use of MEDIA_OBJECT_PRT and MI_ARB_ON_OFF are optional.

- MEDIA_VFE_STATE
- MEDIA_INTERFACE_DESCRIPTOR_LOAD
- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 1)
- MEDIA_OBJECT_PRT (with VFE_STATE_FLUSH set and PRT NEEDED set)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF) // Arbitration must be turned off while sending objects for group 1
- Several MEDIA_OBJECT command (for barrier group 1)
- MI_ARB_ON_OFF (ON) // Arbitration is allowed
- MEDIA_STATE_FLUSH (optional, only if barrier dependency is needed)
- MEDIA_INTERFACE_DESCRIPTOR_LOAD (optional)
- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 2)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF) // Arbitration must be turned off while sending objects for group 2
- Several MEDIA_OBJECT command (for barrier group 2)
- MI_ARB_ON_OFF (ON) // Arbitration is allowed
- ...
- MI_FLUSH

Commands for the GPGPU pipe (GPGPU_OBJECT and GPGPU_WALKER) should be separated from commands for the Media pipe (MEDIA_OBJECT*) by an MI_FLUSH.

Parameterized Media Walker

The Parameterized Media Walker is a hardware thread generation mechanism that creates threads associated with units in a generalized 2-dimensional space, for example, blocks in a 2D image. With a small number of unit step vectors, the walker can implement a large number of walking patterns as described hereafter. This command may provide functions that are normally handled by the host software, thus, may be used to simplify the host software and GPU interface.

The walker described herein is doubly nested, where essentially a “local” walker can perform a variety of 2-dimensional walking patterns and a “global” walker can perform similar 2-dimensional walking

patterns upon many local walkers. The local walker has 3 levels (outer, middle, and inner) while the global walker has 2 levels (outer and inner). Thus, the algorithm has 5-nested loops that modify local state based on user-defined unit step vectors.

The Walker's programmability is derived from:

- The walker traverses a unit-normalized surface. Some example unit sizes:
    - 1x1: Walking pixels
    - 4x4: Walking sub-blocks
    - 16x16: Walking macro-blocks
    - 32x16: Walking macro-block-pairs
- The use of 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
- And a small number of special mode controls for the inner-most loop which are aimed at efficiently dividing 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,
- Inner unit vector

The local inner loop has some special modes that will be described later. A table of the user inputs and some example values are given 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

It should be emphasized that the value of what a "unit" represents is implicitly defined by the user. In other words, the walker traverses a "unit normalized space" that is not inherently bound to pixel walking.

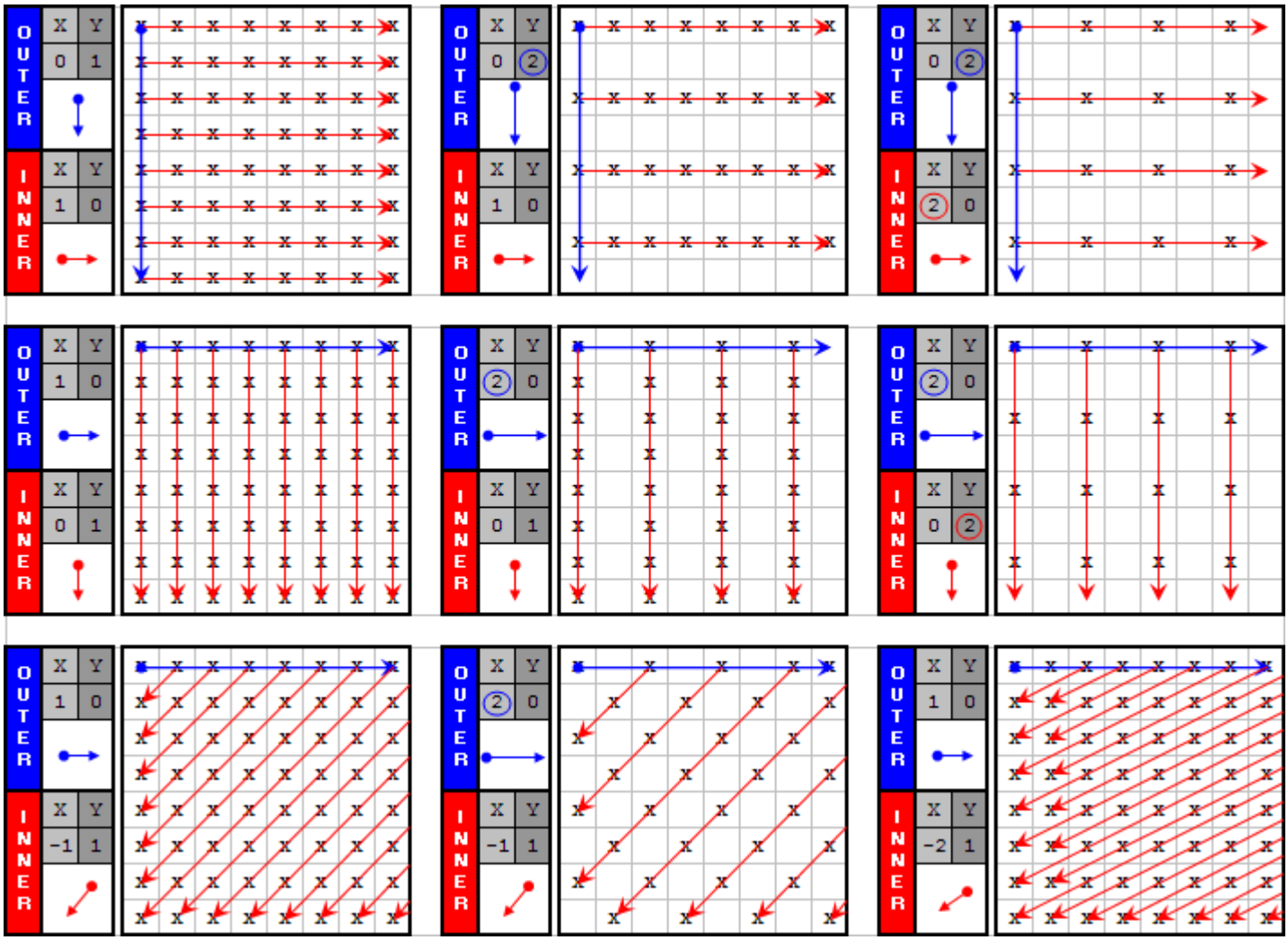
If the smallest unit of work the user wants to walk is a 4x3 block of pixels, you can program the inner loop to step (4,3) or (1,1):

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

It should be noted that hardware doesn't contain enough bits for pixel walking for pixel resolution like 1920x1088. The intended usage of the walker is for block walking whereas the block size is not relevant to the walker parameters.

Basic Parameters for the Local Loop

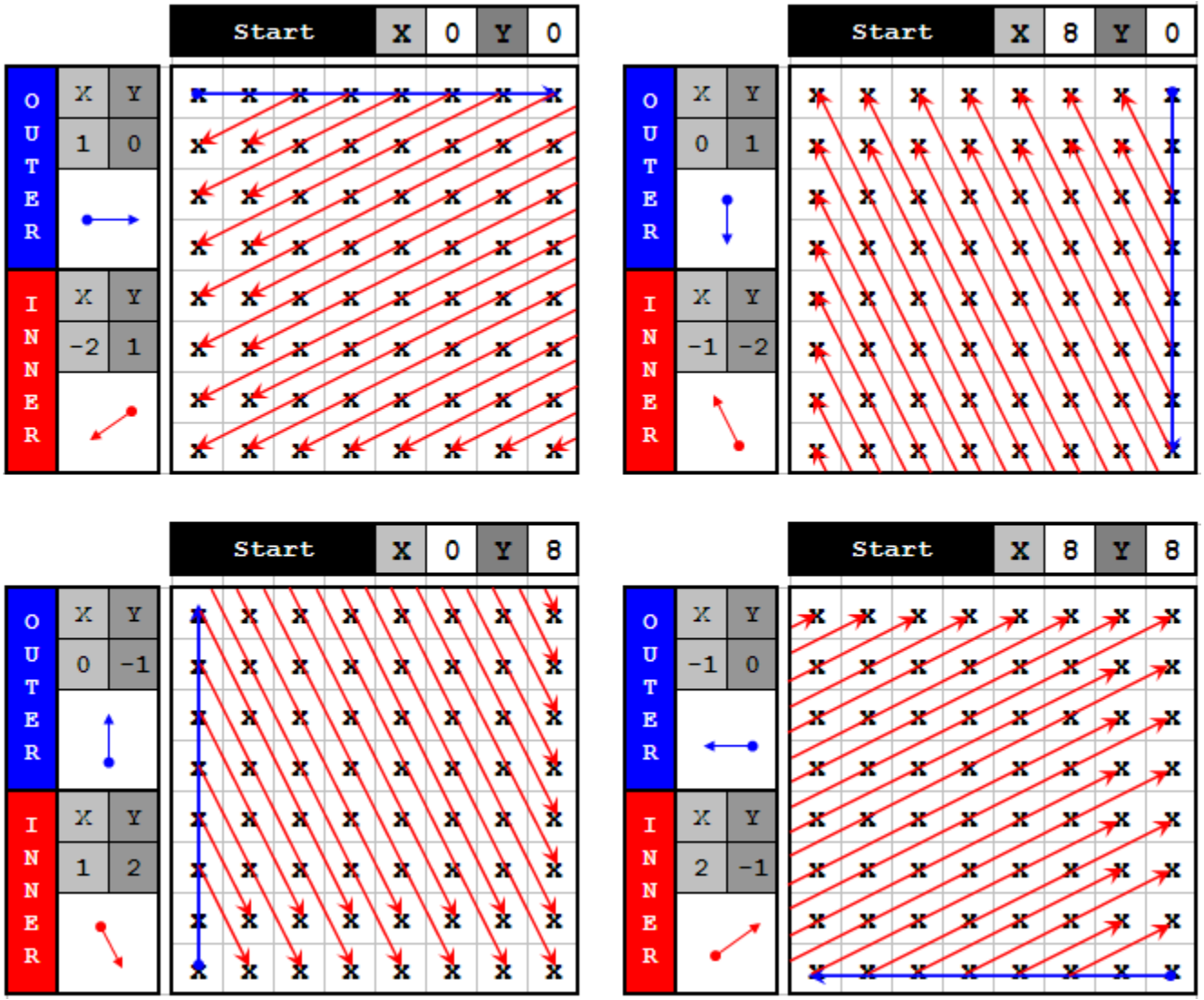
The local inner and outer loop xy-pair parameters alone can describe a large variety of primitive walking patterns. Below are 9 primitive walking patterns generated by varying only the inner and outer unit step vectors of the local loop:



- The top row shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Rows and columns can easily be skipped by increasing the unit step vectors above one.

- The middle row the outer unit vector pointing right (+X) and the inner unit vector pointing down (+Y). Again, rows and columns are skipped by increasing the unit step vectors beyond one.
- The last row shows the capability to walk angles not perpendicular to the edge. The 1st shows a 45° walking pattern by setting the inner unit vector to (-1,1). The 2nd shows a checkerboard pattern by skipping every other outer loop and retaining the inner unit vector of (-1,1). The 3rd shows a 26.5° walking pattern by setting the inner unit vector to (-2,1).

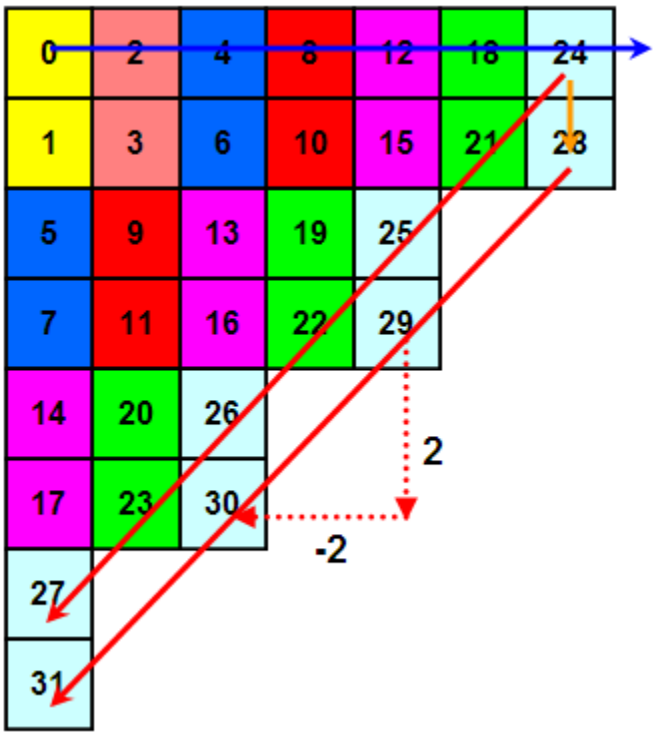
The block resolution, shown as [8,8], and the starting location, currently [0,0], can be varied and the above patterns can be stretched and rotated many ways. The diagram below shows an example of where the start position and unit step vectors can be set to achieve a full rotation of the same pattern:



MbAff-Like Special Case in Local Loop

The local loop has an additional middle loop that is used to achieve some specific walking patterns, with MBAFF mode especially in mind. A pattern to handle MBAFF AVC content is to walk the top macroblocks

of all macroblock pairs (MB-pairs) on a wavefront followed by the respective bottom macroblocks. The pattern is shown below.

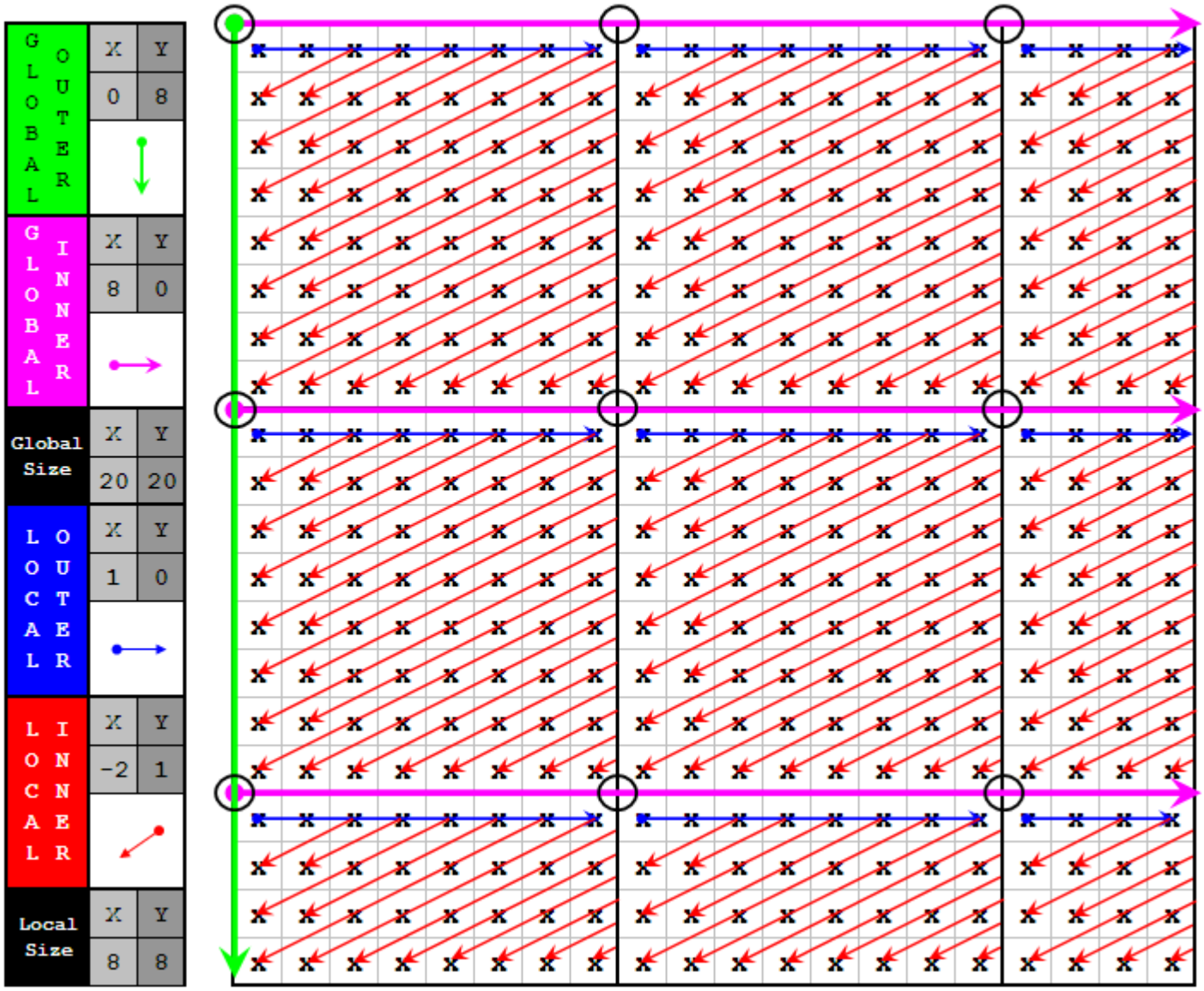


The outer loop unit step vector would be [1, 0] and the inner loop unit step vector would be [-2, 2]. A third loop is necessary to repeat the inner loop, only shifted down a unit before restarting. Thus, a middle loop with a unit step vector of [0,1] would achieve this MBAFF pattern. Additionally, the number of "extra steps" taken by the middle loop would be 1 in this case.

The addition of a middle loop also creates more overall flexibility, which seems necessary due to the integer-based unit step vector solution proposed (Manhattan distance issues etc.).

Global Loop

The same set of general parameters is used to describe the global loop as well. Thus, a global loop that is walking a raster-scan pattern can be combined with a local loop that is walking a 26.5° pattern (or vice-versa). As shown in the example below, if the local block size [8,8] is not an even multiple of the global resolution [20,20], the slack is still processed by dynamically changing the local block resolution.



The global loop will always resolve to be the upper-left corner of the local loop, shown above black circles. Note that local loop can still start in any corner of the local block, but the local (0,0) will always be the location where global loop begins the local loop, hence the upper-left corner.

The user can specify the starting location of the global loop as with the local loop. If the user were to set the global starting location to (16,16) in the previous example, after inverting the global outer and global inner unit step vectors the same pattern would be achieved in the reverse order. Note that the slack would still be handled along the right and bottom edge of the global image in that case. The user could have also started at (12,12) in which case the slack would be handled on the left and top faces.

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
```

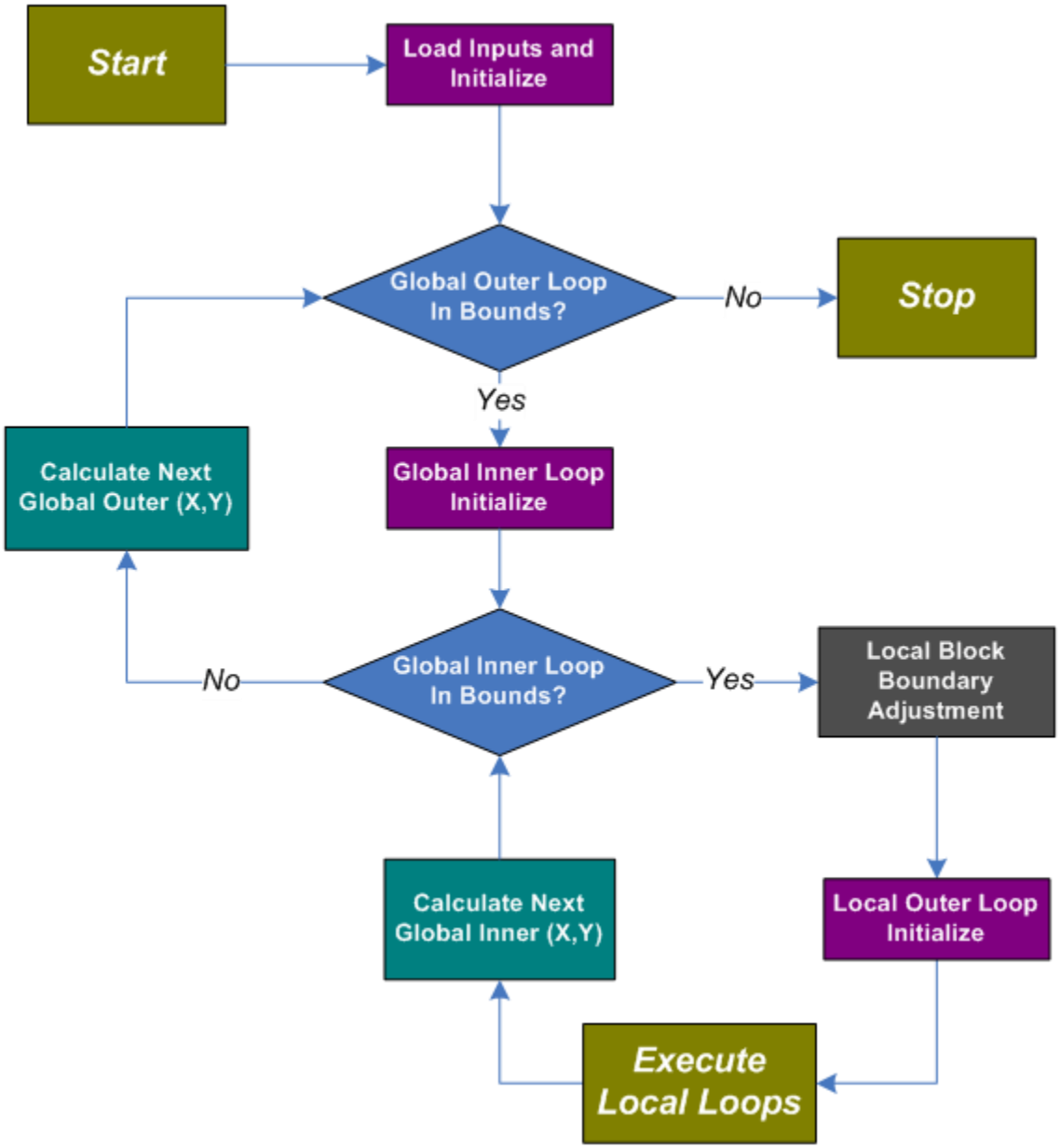
The pseudo-code has the following characteristics:

- There are 5 levels of iteration
- The highest 2 levels are called “global” and the lowest 3 levels are called “local”
  - The global loop is split into an outer and an inner loop.
  - The local loop is split into an outer, a middle, and an inner loop.
  - A bounding box for the global and local resolution is defined by the user.
  - The starting location within each bounding box is also specified by the user.
- Each of the 5 loops has its own persistent
  - 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 given level can be calculated while the next lower level is still executing. Additionally, the result can be used to check to see if the current level will execute again once control is returned.

The flow of the global outer and inner loops is:

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 condition, calculate the next position for the current loop level and repeat.

Walker algorithm flowchart for the Global Loop



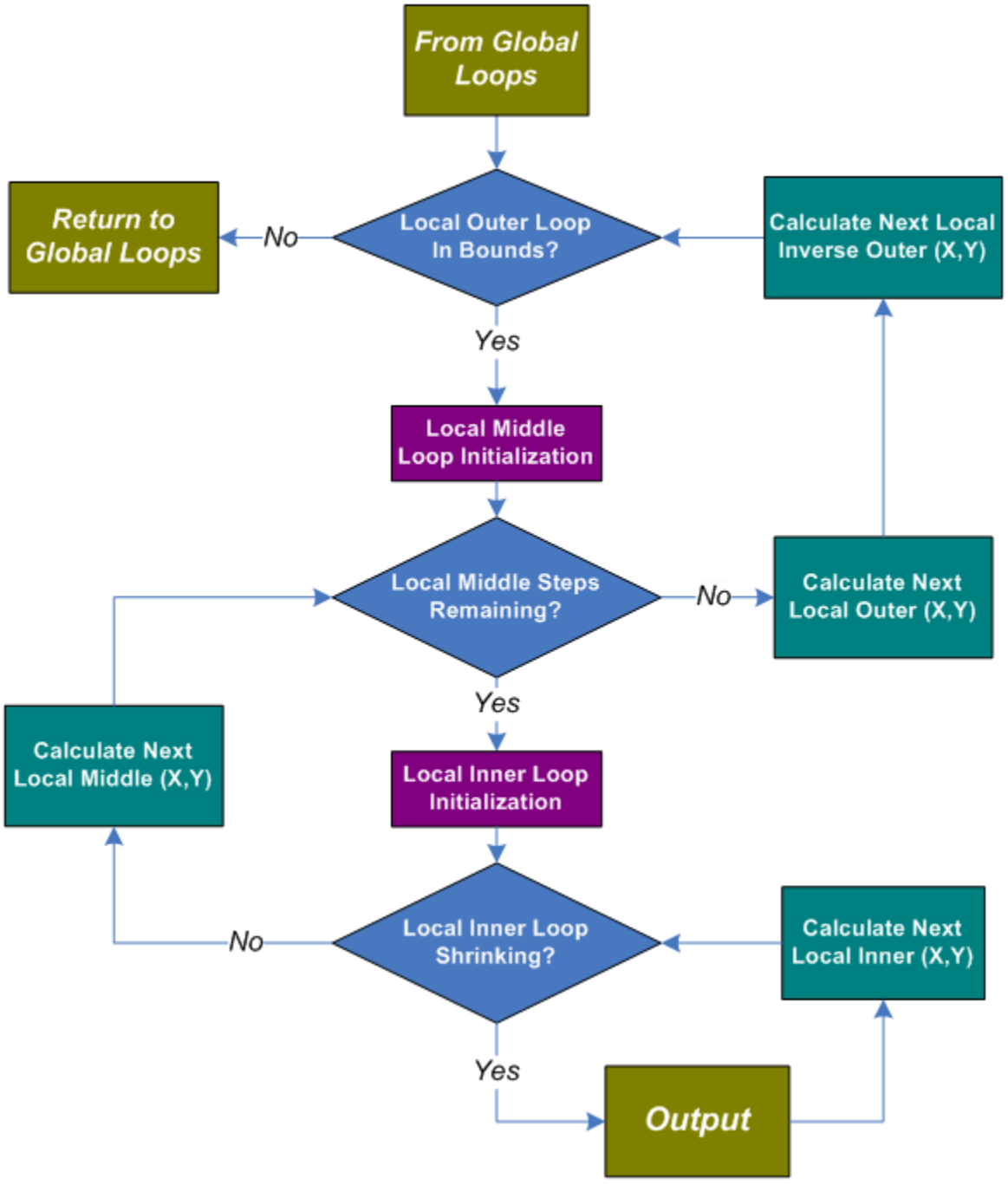
Take note of the grey box "Local Block Boundary Adjustment". This logic is necessary to adjust the local block size when the distance between the current global position to the edge of the image is less than the local resolution. Additionally, the local starting positions might be modified here as well if the defined starting position is larger than the new local block size.

The flow of the 3 local loops does not vary much from the 2 global loops. The differences are:

- In addition to a boundary check, the local middle loop 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 if the prior distance between the x,y starting and ending points are greater than their current distance. If this is true, it implies that 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



Barriers and Shared Local Memory

Barriers and Shared Local Memory can provide advantages to general media applications. Barriers can be used to synchronize between media threads more efficiently than using atomics, while SLM can be used to share data between tightly associated threads.

Allows Barriers and SLM to be used with a more generalized walker, as well as adding the ability to use the scoreboard at the same time. To implement this there needs to be an identifying number similar to the Thread Group ID which can be used to track and free resources.

For MEDIA_OBJECT_WALKER there are already numbers available which can be used as the group id – the various loop counts that the walker maintains. To provide flexibility, the programmer will be allowed to specify which loop counts will form the group id and which will count the threads inside each group. If cross-slice barriers are disabled (see section 1.7), the walker will be required to 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. The programmer is responsible to ensure that the number of threads generated per group is not larger than the threads available in a subslice.

MEDIA_OBJECT_WALKER adds the X/Y values for the various loops added together to produce a single X/Y. Since walking patterns can be produced which have overlapping X/Y values, this can't be used for the global id, instead the execution counts (a count of how many times each loop is executed) are concatenated together to produce an id number. The GPGPU thread group id is a 96-bit number, so the 49 bits created by all the execution counts will fit easily.

The media walker has 5 nested loops for producing the X/Y; in addition there is an innermost color loop:

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

The bits to use as the global id are specified with a parameter which specifies at which point to switch between the group id and the per thread id. Unused loops will always have execution counts of 0. The group id is formed from whichever execution counts are enabled in the Group ID Loop Select, with the selected execution counts concatenated into the LSBs of the group id, with any unused MSBs 0.

Example: if a MEDIA_OBJECT_WALKER specifies that the Outer local loop count and above will form the group id, then every iteration of the color, inner local, and mid local will dispatch a thread with the same group id. TSG will allocate a shared barrier and SLM (depending on what is specified in the Interface Descriptor) and ensure that all the threads go to the same subslice. The programmer ensures that this number of threads will fit on a single subslice, unless only global barriers are used, in which case the number of thread must fit on the system. The group id for this example is formed by cat(outer global exec count, inner global exec count, outer local exec count).

When the group id increments, the TSG will allocate a new barrier and SLM and pick an available subslice.

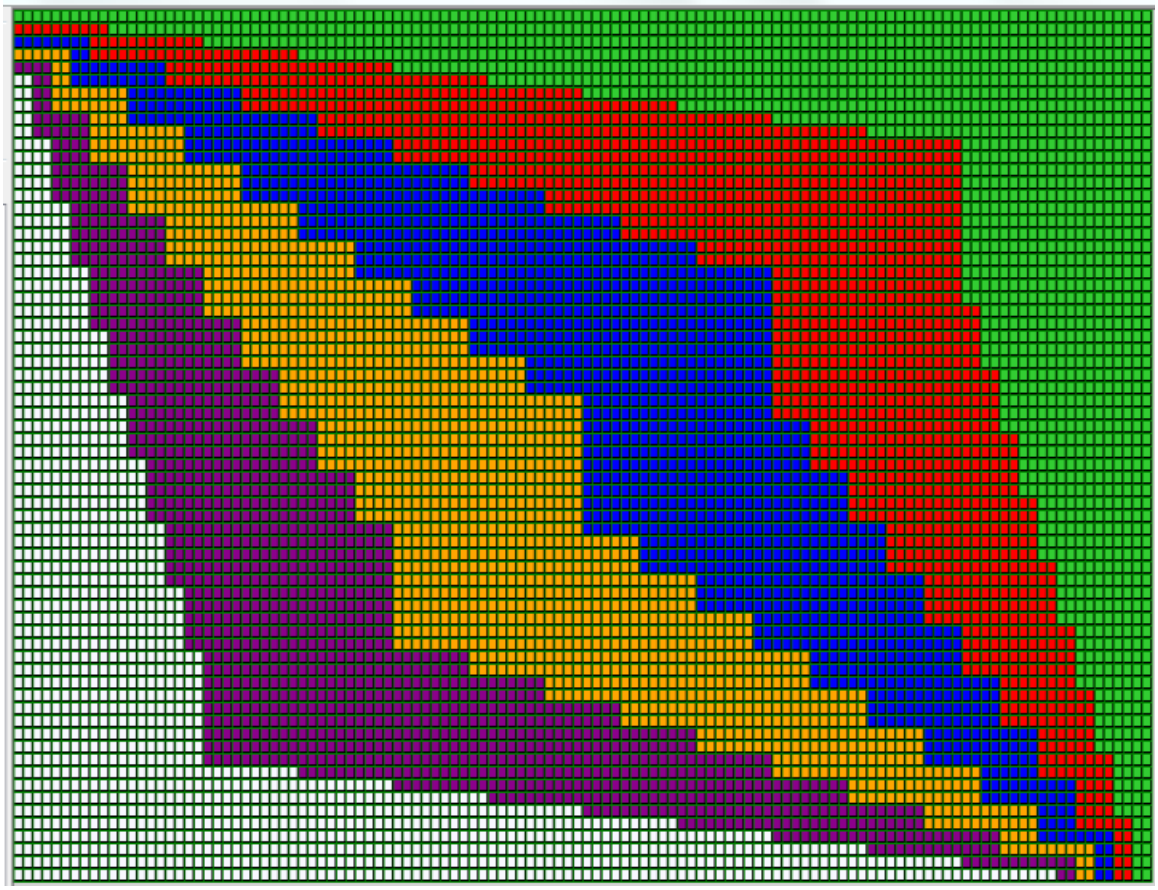
Before the MEDIA_OBJECT_WALKER command with groups there should be a MEDIA_STATE_FLUSH with a watermark bit and matching Interface Descriptor.

Flexible Dispatch of Local Loop

The local loop is automatically split between the availble subslices in such a way as to keep adjacent blocks next to each other to improve cache and execution efficency. The length of a single iteration of the local loop is split into equal segments, one for each subslice that is currently powered up.

The dispatches for the local loop are done such 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.

Example of splitting a wavefront between 6 destinations



Each color indicates a separate segment which is dispatched to a different subslice.

In the example above, each local loop walks a diagnal line from the lower left to the upper right, while the global loop steps between the lines. This is a typical usage model where the dependencies are to the left and above. The first few iterations are in the upper right corner and so have few blocks dispatched

per local loop. Farther down, the length of the local loop gets large enough that the 6 available subslices are full of threads from each segment running in parallel and being dispatched in an even manner.

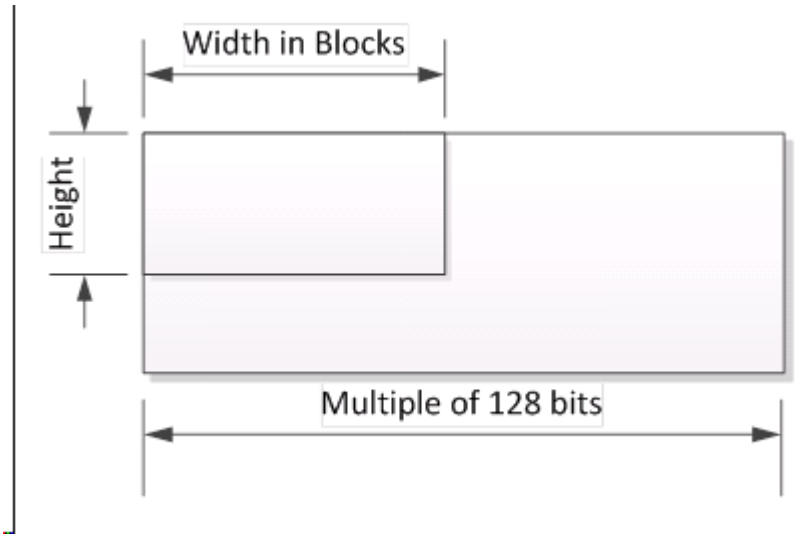
Masked Dispatch

Masked dispatch adds a dispatch mask to the Media_Object_Walker command. This provides a bit per dispatch which determines whether the dispatch is dropped or not. This allows Media_Object_Walker to be used with irregularly shaped areas. When using this mode, the walker should be programmed to walk the surface as a simple rectangle with X changing in the innermost loop.

The dispatch mask is stored in CURBE, which means that a Media_Object_Walker command that uses it cannot use CURBE as part of the dispatch payload. The CURBE is loaded with a bitmask with each bit corresponding to the X/Y blockid (not the pixel X/Y). The byte address of each bit is :

Byte_address = (Y * pitch + X) » 8

Pitch is the number of blocks in the surface horizontally rounded up to the next 128, 256 or 512 bits.



The Media_Object_Walker command allows a maximum 9-bit X and Y, which would correspond to a 512-bit wide x 512 deep surface in CURBE. For 16x16 pixel blocks this would be 8k x 8k pixels.

Note that in this mode the walker will dispatch in sequence rather than splitting the local loop into segments which are dispatched to individual sub-slices (as described in “Flexible Dispatch of Local Loop”.

Scoreboard Control

A hardware mechanism controls the dispatch of root threads. Without using this hardware mechanism, only the dispatch of a SRT is managed by a parent root thread using the SRT message to TS.

There is a scoreboard hardware in TS unit. The scoreboard is addressed by the 18-bit (X, Y) scoreboard field in VFE DWord, where (X, Y) is typically used as the Cartesian coordinate of the working unit in a 2D frame but may be interpolated in other ways. When a root thread is dispatched, the entry at (X, Y) is marked. When the root thread is terminated, the corresponding bit in the scoreboard is cleared.

Each root thread may have up to eight dependencies. The dependency relation is described by the state value of Scoreboard Controls in terms of related distance of (deltaX, deltaY). There is a global scoreboard enabling in the state as well as the-per thread enabling for each dependency.

TS stalls the dispatch of a root thread if any scoreboard entry, which is denoted by (Scoreboard X + deltaX, Scoreboard Y + deltaY), matching with any enabled dependencies is marked as in-flight. The thread is dispatched only after all dependencies are cleared.

For a root thread, TS stalls the dispatch of the thread only if the dependent scoreboard entries of the thread are marked. It does not automatically stalls the dispatch for destination collision if (deltaX = 0, deltaY=0) is not set in the scoreboard state. This kind of scoreboard destination collision is due to the scoreboard wrap-around (or aliasing), which must be avoided. With 9-bit per X, Y field, the hardware scoreboard can support a frame that is subdivided up to 512x512 threads without a scoreboard aliasing.

In addition to the above 'stalling scoreboard', Media Pipe may also support a non-stalling scoreboard. With non-stalling, a thread is dispatched with the dependent threads marked. The thread dependency affects the issuing of a *sendc* instruction. See vol5d Execution Unit ISA for details.

Scoreboard Support in Device Hardware

Device	Stalling scoreboard	Non-Stalling scoreboard
	Yes	Yes

Programming Note	
Context:	Scoreboard Control
• The hardware scoreboard only handles root threads, but not child threads. This limitation may be revisited when future application requirement changes. • The usage of hardware scoreboard and SRT are mutually exclusive. In other words, when hardware scoreboard is used, SRT should not be issued.	

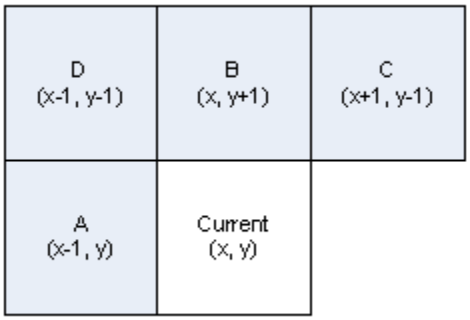
AVC-Style Dependency Example

For AVC decoding, dependencies for a given macroblock may be set based on the availability of neighbor macroblocks, namely A, B, C, D and left-bottom neighbors (left-bottom only if MbAff = 1), as well as the current macroblock's address, MbAff flag and FieldMbFlag. For a macroblock in a progressive frame picture or a field picture, one macroblock may depend on up to four neighbors, A, B, C and D as shown in Neighbor addresses of a macroblock in a progressive frame picture (MbAff = 0) or a field picture with up to 4 dependencies. For a macroblock in a MbAff pair, it may depend on up to three, five or eight neighbors as shown in Neighbor addresses of the first macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies and Neighbor addresses of the second macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies, based on the current macroblock's address and FieldMbFlag.

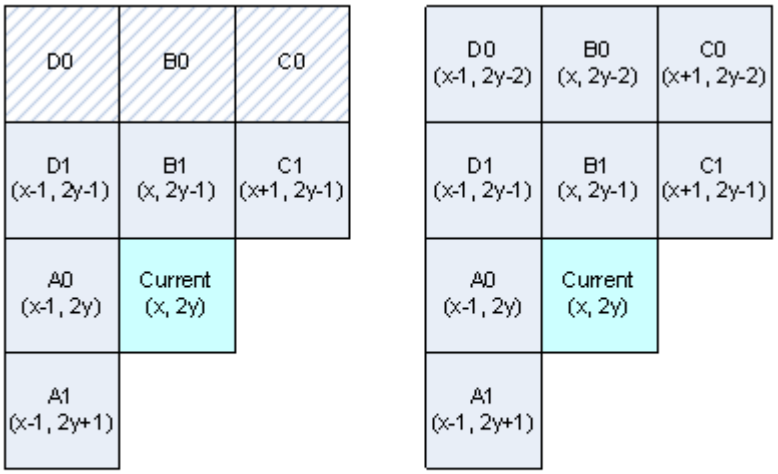
The neighbor's availability depends on the corresponding **IntraPredAvailFlagA|B|C|D|E** flags for the macroblock (or the macroblock pair). Hardware assumes that the flags are set correctly in the

MEDIA_OBJECT_EX command as shown in Macroblock indices for field picture destination. For simplicity, the left neighbor pair (A0 and A1) availability for a MbAff macroblock can be determined as a group by **IntraPredAvailFlagA | IntraPredAvailFlagE**. For the second macroblock in a 'frame' MbAff pair, it depends on the first macroblock in the pair and it is always available.

Neighbor addresses of a macroblock in a progressive frame picture (MbAff = 0) or a field picture with up to 4 dependencies



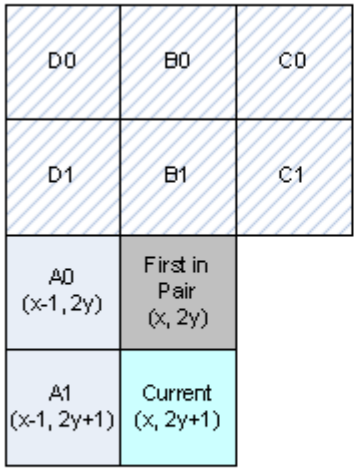
Neighbor addresses of the first macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies



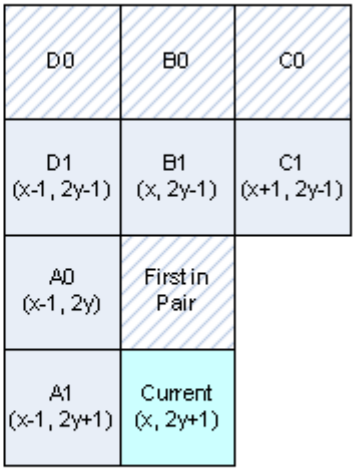
(a) Neighbors for the first macroblock in a 'frame' MbAff pair

(b) Neighbors for the first macroblock in a 'field' MbAff pair

Neighbor addresses of the second macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies



(b) Neighbors for the second macroblock in a 'frame' MbAff pair



(b) Neighbors for the second macroblock in a 'field' MbAff pair

Neighbor Availability

MbAff	FieldMbFlag	VertOrigin[0]	A	B	C	D	LB	Description
0	0/1	0/1	□	□	□	□	•	Progressive or Field picture
1	0	0	□	□	□	□	□	1st Frame MbAff macroblock
1	0	1	□	na	0	na	□	2nd Frame MbAff macroblock
1	1	0	□	□	□	□	□	1st Field MbAff macroblock
1	1	1	□	□	□	□	□	2nd Field MbAff macroblock

Interface Descriptor Selection

In VLD mode, the Interface Descriptor Offset field in the MEDIA_OBJECT command is ignored by hardware. Instead, the interface descriptor offset is computed by hardware based on the decoded macroblock parameters and a remapping table.

First a macroblock index is computed based on parameters such as picture structure, motion type, prediction type, DCT type, intra-coding type and motion vector present information. *Interface Descriptor Selection* provides the macroblock index table for a frame-picture destination buffer (with Picture Structure = 11). *Interface Descriptor Selection* shows macroblock indices for a field-picture destination

buffer (with Picture Structure = 01 or 10). As Picture Structure is a state variable that is not changed until a pipeline flush, the macroblock indices can be computed separately for different Picture Structure.

After the macroblock index is computed, it is used as the index into the Interface Descriptor Remap Table to derive the final interface descriptor offset value. The Interface Descriptor Remap Table is provided as part of the VLD state.

The interface descriptor offset value multiplied by the interface descriptor size is then added to the interface descriptor base pointer to generate the interface descriptor pointer for the post-VLD thread.

The last three columns in *Interface Descriptor Selection* and *Interface Descriptor Selection* indicate whether a macroblock index is applicable for a given Picture Coding type (I, P or B). A 'Y' (or a 'N') means the macroblock index on the row is valid (or invalid) for the Picture Type shown on the column. Taking a frame picture destination for example, only macroblock indices 0 and 8 are valid for an I-picture; indices 0-3 and 8-11 are valid for a P-picture; and for a B-picture, only indices 3 and 11 are not valid.

Developers can use the remap table for kernel development to fine-tune system performance and reduce software complexity. For example, if the destination is a frame picture, the kernel for a macroblock with dual-prime motion in a P-picture (macroblock index = 3) may be identical to that for a macroblock with bidirection field motion in a B picture (macroblock index = 7). A common set of interface descriptors can be configured once for frame picture destinations, and reused without change when the destination is of I-, P- and B- picture coding type.

In another case, if it is determined that kernel software is responsible for handling DCT types for a frame picture destination, then macroblock index i and i+8 , for i = 0 to 7, can be mapped to the same interface descriptor.

Macroblock Indices for Frame Picture Destination

Macroblock Index	Interface Descriptor Kernel Function (Frame Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward frame motion	N	Y	Y
2	Forward field motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward frame motion	N	N	Y
5	Backward field motion	N	N	Y
6	Bidirectional frame motion	N	N	Y
7	Bidirectional field motion	N	N	Y
8	I macroblock with field DCT	Y	Y	Y
9	Forward frame motion with field DCT	N	Y	Y
10	Forward field motion with field DCT	N	Y	Y
11	P picture, dual-prime motion with field DCT	N	Y	N
12	Backward frame motion with field DCT	N	N	Y

Macroblock Index	Interface Descriptor Kernel Function (Frame Picture Destination)	I	P	B
13	Backward field motion with field DCT	N	N	Y
14	Bidirectional frame motion with field DCT	N	N	Y
15	Bidirectional field motion with field DCT	N	N	Y

Macroblock Indices for Field Picture Destination

Macroblock Index	Interface Descriptor Kernel Function (Field Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward field motion	N	Y	Y
2	Forward 16x8 motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward field motion	N	N	Y
5	Backward 16x8 motion	N	N	Y
6	Bidirectional field motion	N	N	Y
7	Bidirectional 16x8 motion	N	N	Y

VC1-Style Dependency Example

For VC1, only one dependency may be set depending on the availability of the upper neighbor macroblock.

Macroblock sequence order in a VC-1 picture with WidthInMblk = 5 and HeightInMblk = 6

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24
5	25	26	27	28	29

Multiple Slice Considerations

For products with multiple slices, the Render Cache is separate per slice with no hardware coherency. This means that the programmer must ensure coherency by one of these methods:

- Using write commit when writing to the Render Cache.
- Using Data Cache instead of the Render Cache.

- Different slices only access separate cache lines. using a hashing algorithm combined with the slice select bits of the MEDIA_OBJECT/GPGPU_OBJECT commands.

Interrupt Latency

Command Streamer is capable of context switching between primitive commands.

For all independent threads, it is not much a problem. The interrupt latency is dictated by the longest command that is likely to have the largest number of threads. For VLD mode, such a command may be corresponding to a largest slice in a high definition video frame. This is application dependent, there are not much host software can do. For Generic mode, programmer should consider to constrain the compute workload size of each thread.

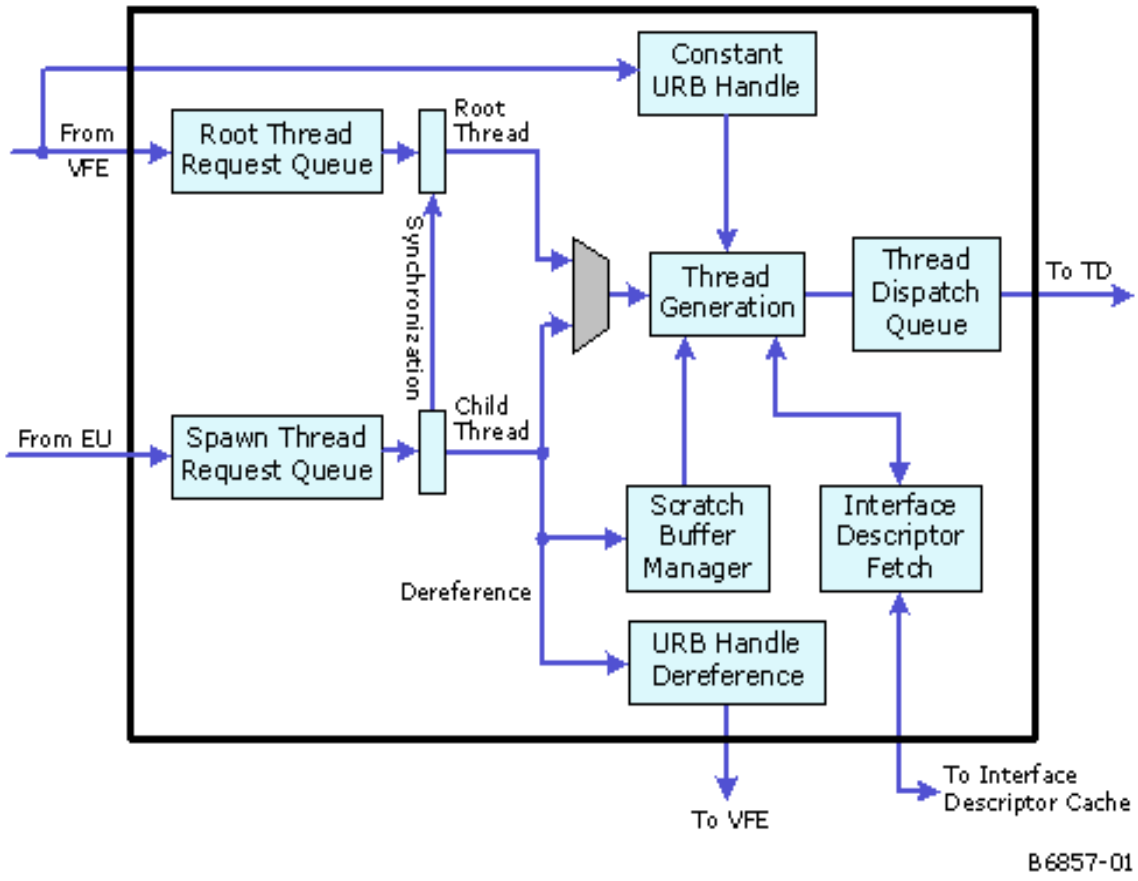
In modes with child threads, a root thread may persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. The Software designer should partition the workload to restrict the duration of each root thread. For example, this may be achieved by partitioning a video frame and assigning separate primitive commands for different data partitions.

In modes with synchronized root threads, a synchronized root thread is dependent on a previous root or child thread. This means context switch is not allowed between the primitive command for the synchronized root thread and the one for the depending thread. So no command queue arbitration should be allowed between them. Software designer should also restrict the duration of such non-interruptible primitive command segments.

Thread Spawner Unit

The Thread Spawner (TS) unit is responsible for making thread requests (root and child) to the Thread Dispatcher, managing scratch memory, maintaining outstanding root thread counts, and monitoring the termination of threads.

Thread Spawner block diagram



Root Threads and Child Threads

Thread requests sourced from VFE are called **root threads**. These threads may be creating subsequent child threads.

Root Threads

A root thread may be a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode. Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.

Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
    - Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.
2. TS then builds the transparent header and the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.
    - It is the root thread's responsibility (software) to guarantee that all its children have retired before the root thread can retire.

URB Handles

VFE is in charge of allocating URB handles for root threads. One URB handle is assigned to each root thread. The handle is used for the payload into the root thread.

Children Present is a command variable in the _OBJECT command.

If Children Present is not set (root-without-child case), TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched.

If Children Present is set (root-with-child case), the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.

Root to Child Responsibilities

Any thread created by another thread running in an EU is called a **child thread**. Child threads can create additional threads, all under the tree of a root which was requested via the VFE path.

A root thread is responsible of managing pre-allocated resources such as URB space and scratch space for its direct and indirect child threads. For example, a root thread may split its URB space into sections. It can use one section for delivering payload to one child thread as well as forwarding the section to the child thread to be used as return URB space. The child thread may further subdivide the URB section into subsections and use these subsections for its own child threads. Such process may be iterated. Similarly, a root thread may split its scratch memory space into sections and give one scratch section for one child thread.

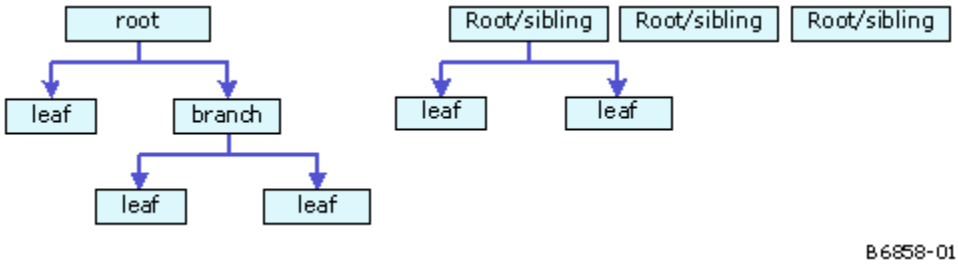
TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

Multiple Simultaneous Roots

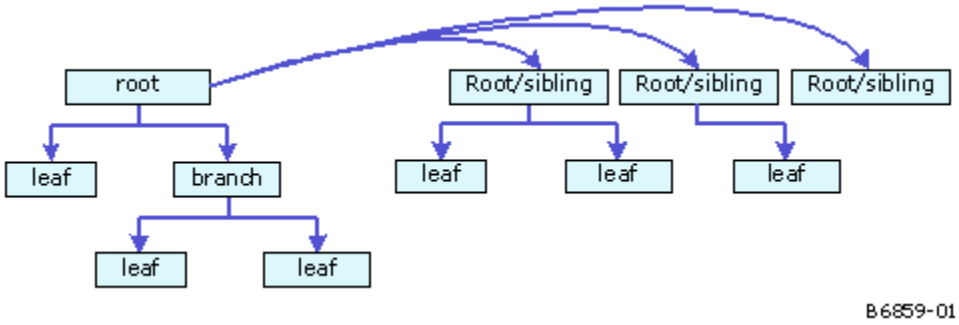
Multiple root threads are allowed concurrently running in GEN4 execution units. As there is only one scratch space state variable shared for all root threads, all concurrent root thread requiring scratch space share the same scratch memory size. *Multiple Simultaneous Roots* depicts two examples of thread-thread relationship. The left graph shows one single tree structure. This tree starts with a single root thread that generates many child threads. Some child threads may create subsequent child threads. The right graph shows a case with multiple disconnected trees. It has multiple root threads, showing sibling roots of disconnected trees. Some roots may have child threads (branches and leafs) and some may not.

There is another case (as shown in *Multiple Simultaneous Roots*) where multiple trees may be connected. If a root is a synchronized root thread, it may be dependent on a preceding sibling root thread or on a child thread.

Examples of thread relationship



A example of thread relationship with root sibling dependency



Synchronized Root Threads

A synchronized root thread (SRT) originates from a MEDIA_OBJECT command with Thread Synchronization field set. Synchronized root threads share the same root thread request queue with the non-synchronized roots. A SRT is not automatically dispatched. Instead, it stays in the root thread request queue until a spawn-root message is at the head of the child thread request queue. Conversely, a spawn-root message in the child thread request queue will block the child thread request queue until the head of root thread request queue is a SRT. When they are both at the head of queues, they are taken out from the queue at the same time.

A spawn-root message may be issued by a root thread or a child thread. There is no restriction. However, the number of spawn-root messages and the number of SRT must be identical between state changes. Otherwise, there can be a deadlock. Furthermore, as both requests are blocking, synchronized root threads must be used carefully to avoid deadlock.

When Scoreboard Control is enabled, the dispatch of a SRT originated from a MEDIA_OBJECT_EX command is still managed by the same way in addition to the hardware scoreboard control.

Deadlock Prevention

Root threads must control deadlock within their own child set. Each root is given a set of preallocated URB space; to prevent deadlock it must make sure that all the URB space is not allocated to intermediate children who must create more children before they can exit.

There are limits to the number of concurrent threads. The upper bound is determined by the number of execution units and the number of threads per EU. The actual upper bound on number of concurrent threads may be smaller if the GRF requirement is large. Deadlock may occur if a root or intermediate parent cannot exit until it has started its children but there is no space (for example, available thread slot in execution units) for its children to start.

To prevent deadlock, the maximum number of root threads is provided in VFE state. The Thread Spawner keeps track of how many roots have been spawned and prevents new roots if the maximum has been reached. When child threads are present, it is software's responsibility to constrain child thread generation, particularly the generation of child threads that may also spawn more child threads.

Child thread dispatch queue in TS is another resource that needs to be considered in preventing deadlock. The child thread dispatch queue in TS is used for (1) message to spawn a child thread, (2) message to spawn a synchronized root thread, and (3) thread termination message. If this queue is full, it will prevent any thread to terminate, causing deadlock.

For example, if an application only has one root thread (max # of root threads is programmed to be one). This root thread spawns child threads. In order to avoid deadlock, the maximum number of outstanding child thread that this root thread can spawn is the sum of the maximum available thread slots plus the depth of the child thread dispatch queue minus one.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1)

Adding other root threads (synchronized and/or non-synchronized) to the above example, the situation is more complicated. A conservative measure may have to use to prevent deadlock. For example, the root thread spawning child threads may have to exclude the max number of root threads as in the following equation to compute the maximum number of outstanding child threads to be dispatched.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1) – (Max Root Threads-1)

Child Thread Life Cycle

When a (parent) thread creates a child thread, the parent thread behaves like a fixed function. It provides all necessary information to start the child thread, by assembling the payload in URB (including R0 header) and then sending a spawn thread message to TS with following data:

- An interface descriptor pointer for the child thread.
- A pointer for URB data

The interface descriptor for a child may be different from the parent – how the parent determines the child interface descriptor is up to the parent, but it must be one from the interface descriptor array on the same interface descriptor base address.

The URB pointer is not the same as a URB handle. It does not have an URB handle number and does not appear in any handle table. This is acceptable because the URB space is never reclaimed by TS after a child is dispatched, but rather when the parent releases its original handles and/or retires.

Programming Note	
Context:	Child Thread Life Cycle
The child request is stored in the child thread queue. The depth of the queue is limited to 8, overrun is prevented by the message bus arbiter which controls the message bus. The arbiter knows the depth of the queue and will only allow 8 requests to be outstanding until the TS signals an entry has been removed.	

As mentioned previously, child threads have higher priority over root threads. Once TS selects a child thread to dispatch, it follows these steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
2. TS then builds the transparent header but not the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. Once the dispatch is done, TS can forget the child – unlike roots, no bookkeeping is done that has to be updated when the child retires.

If more data needs to be transferred between a parent thread and its child thread than that can fit in a single URB payload, extra data must be communicated via shared memory through data port.

Arbitration between Root and Child Threads

When both root thread queue and child thread queue are both non-empty, TS serves the child thread queue. In other words, child threads have higher priority over root threads. The only condition that the child thread queue is stalled by the root thread queue is that the head of child thread queue is a root-synchronization message and the head of root thread queue is not a synchronized root thread.

Persistent Root Thread

A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread, and only one PRT is allowed in the system at a time.

On a context switch interrupt, instead of proceeding to completion, a PRT can save its software context and terminate. The PRT can be restarted later, even if it had completed normally the last time it was executed. Therefore, the PRT must always save enough context (via data port messages to a predefined surface) to allow it to restart from where it left off (including determining that it has nothing left to do).

Because only one PRT can execute at a time, once the next PRT starts, the previous one will never be restarted, thus the context save surface can be reused from one PRT to the next.

A PRT may check the Thread Restart Enable bit in the R0 header to find out whether it is a fresh start or resumed from a previous interrupt and then can continue operations from that previously saved context.

A PRT can be interleaved with other root (such as parent root thread, or synchronized root thread) and child threads. A parent root thread is not necessarily a PRT, and doesn't have to be as long as it can be finished in deterministic time that is shorter than required for fine-grain context switch interrupt.

Use of PRT must follow the following rule:

- There can only be one PRT in the media pipeline at a given time. That means, there shall not be any other media primitive commands (MEDIA_OBJECT or MEDIA_OBJECT_EX) between it and the previous MI_FLUSH command. In other words, when multiple such PRTs are used in a sequence of media primitive commands, MI_FLUSH must be inserted.

Media State Model

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

All Interface Descriptors have the same size and are organized as a contiguous array in memory. They can be selected by Interface Descriptor Index for a given kernel. This allows different kinds of kernels to coexist in the system.

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

Media State and Primitive Commands

This section contains various commands for media, all with the RenderCS source.

MEDIA_VFE_STATE

MEDIA_CURBE_LOAD

MEDIA_INTERFACE_DESCRIPTOR_LOAD

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address:

INTERFACE_DESCRIPTOR_DATA

Programming Restriction: Back to back interface descriptor load commands, must have a VFE State command inserted between them.

MEDIA_STATE_FLUSH

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline. It supports loading of inline data as well as indirect data. At least one form of payload (either inline, indirect, or CURBE) must be sent with the MEDIA_OBJECT.

MEDIA_OBJECT

MEDIA_OBJECT_PRT

MEDIA_OBJECT_GRPID

The MEDIA_OBJECT_WALKER command uses the hardware walker in VFE for generating threads associated with a rectangular shaped object. It only supports loading of inline data or CURBE but not indirect data. At least one form of payload must be sent. Control of scoreboards (up to 8) is implicit based on the (X, Y) address of the generated thread and the scoreboard control state.

The command can be used only in Generic modes.

When **Use Scoreboard** field is set, the (X, Y) address and the Color field of the generated thread are used in the hardware scoreboard and the thread dependencies are set by states from the MEDIA_VFE_STATE command.

One or more threads may be generated by this command. This command does not support indirect object load. When inline data is present, it is repeated for all threads it generates. Unlike CURBE, which requires pipeline flush for change, continued change of this kind of 'global' (in the sense of shared by multiple threads from this command) data is supported when MEDIA_OBJECT_WALKER commands are issued without a pipeline flush in between.

MEDIA_OBJECT_WALKER

Media State and Primitive Command Workarounds

Media State and Primitive Commands have some subtle programming restrictions and workarounds, as listed below.

Programming Note
The MEDIA_STATE_FLUSH command is updated to optionally specify all the resources required for the next thread group via an interface descriptor – if the resources are not available the group cannot start.
When a context switches between using the Stalling Scoreboard and the Non-stalling Scoreboard with the Scoreboard Type field in MEDIA_VFE_STATE a stalling PIPE_CONTROL must be placed before the MEDIA_VFE_STATE command.

Media Payload

There are 3 types of payload that the media kernels can have, but not all of them are allowed. The following table lists the legal combinations:

Workload	Commands	Data Stored
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 using Barrier/SLM	Media_Object_GRPID	CURBE
	Media_Object_GRPID	INDIRECT
	Media_Object_GRPID	INLINE
	Media_Object_Walker (with group id)	CURBE

Workload	Commands	Data Stored
	Media_Object_Walker (with group id)	INLINE

For legacy media the payload is simple - each media_object or each dispatch in media_object_walker gets all the payload specified.

For media using barriers/SLM the CURBE and INDIRECT payload is similar to the GPGPU payload method where the payload is split into per-thread and cross-thread sections and each thread in a group gets a the cross-thread and a unique section of per-thread payload. The payload is only split between the threads in a group, and each group gets a duplicate version of the payload.

Media Messages

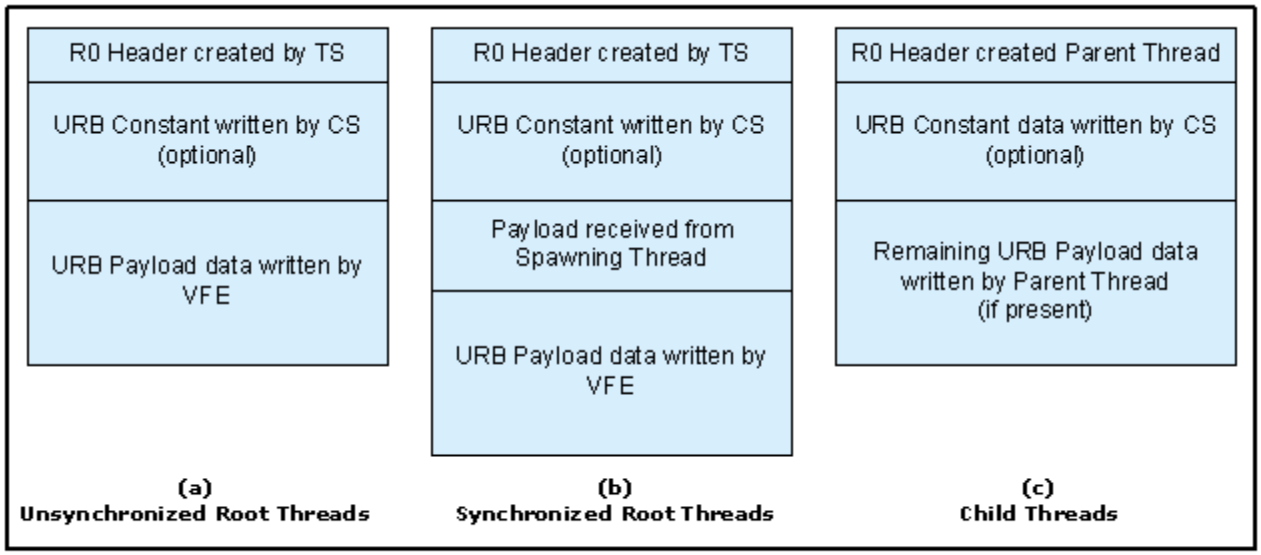
All message formats are given in terms of dwords (32 bits) using the following conventions:

- Dispatch Messages: **R**p.d
- SEND Instruction Messages: **M**p.d

Thread Payload Messages

The root thread's register contents differ from that of child threads, as shown in *Thread Payload Messages*. The register contents for a synchronized root thread (also referred to as 'spawned root thread') and an unsynchronized one are also different. Whether the URB Constant data field is present or not is determined by the interface descriptor of a given thread. This applies to both root and child threads. When URB Constant data field is present for a synchronized root thread, URB constant data field is before the data field received from the spawning thread, which is also before the URB payload data.

Thread payload message formats for root and child threads



Generic Mode Root Thread

The following table shows the R0 register contents for a Generic mode root thread, which is generated by TS. The remaining payloads are application dependent.

R0 Header of a Generic Mode Root Thread

DWord	Bits	Description
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
R0.4	31:5	**Binding Table Pointer.** 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: MBZ
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.** The amount of scratch space, in 1K-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	31:28	Reserved: MBZ
	27:16	Reserved: MBZ
	15:10	Reserved: MBZ
	9:4	**Interface Descriptor Offset.** The offset from the interface descriptor base pointer to the interface descriptor that applies to this object, in units of interface descriptors. Format = U6
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX): This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
R0.1	31:28	Reserved: MBZ
	26:16	**Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U11
	15:11	Reserved: MBZ
	10:0	**Scoreboard X** This field provides the X term of the scoreboard value of the current thread. Format = U11
R0.0	31:24	Reserved: MBZ

DWord	Bits	Description
	23:16	**Scoreboard Mask.** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. **Bit n (for n = 0...7):** Scoreboard n is dependent, where bit 24 maps to n = 0. Format = TRUE/FALSE
	15:0	**URB Handle.** This is the URB handle indicating the URB space for use by the root thread and its children.

Root Thread from MEDIA_OBJECT_PRT

The root thread payload message for a MEDIA_OBJECT_PRT command has a fixed format independent of the VFE mode (e.g. Generic mode or AVC-IT mode). One example GRF register location is given for the condition that CURBE is disabled.

Root Thread Payload Layout for a MEDIA_OBJECT_PRT Command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	N/A	**Constants from CURBE when CURBE is enabled** m is a non-negative value.
R(m+1)	R1	**In-line Data block**

The R0 header field is as the following, which is the same as in other modes except the Thread Restart Enable bit (bit 0 of R0.2).

R0 Header of the Thread Payload of a MEDIA_OBJECT_PRT Command

DWord	Bit	Description
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1K-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved: MBZ
	7:0	**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.
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: MBZ
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 1K-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.

DWord	Bit	Description
		Format = U4 Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two
R0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers. Format = GeneralStateOffset[31:4]
	3:1	Reserved: MBZ
	0	**Thread Restart Enable.** If set, indicates that the persistent root thread (PRT) is being restarted, and context should be restored from the context save area before executing. Format = Enable
R0.1	31:0	Reserved: MBZ
R0.0	31:16	Reserved: MBZ
	15:0	**URB Handle.** This is the URB handle indicating the URB space for use by the root thread and its children.

The inline data block field is the same as in the MEDIA_OBJECT_EX command with zero-filled partial GRF.

Root Thread from MEDIA_OBJECT_WALKER

The root thread payload message for an MEDIA_OBJECT_WALKER command, which must be in Generic mode, has the same format as that of the generic mode root thread format.

Root thread payload layout for a MEDIA_OBJECT_WALKER command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	n/a	**Constants from CURBE when CURBE is enabled** m is a non-negative value
R(m+1)	R1	**In-line Data block**

The R0 header field is identical to that of Generic Mode Root Thread.

The inline data block field is the same as in the MEDIA_OBJECT command with zero-filled partial GRF.

There is no indirect data block field.

MEDIA_OBJECT_GRPID and MEDIA_OBJECT_WALKER with Groups Payload

The RO header of the MEDIA_OBJECT_GRPID & MEDIA_OBJECT_WALKER Payload with groups enabled:

DWord	Bits	Description
R0.7	31:0	**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: 0: No groups, field is 0. 1: cat(InnerGlobalCnt[6:0], OuterLocalCnt[9:0], MidLocalCnt[4:0], InnerLocalCnt[9:0]); rest of group

DWord	Bits	Description
		id is in R0.2. 2: cat(OuterGlobalCnt[6:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0], MidLocalCnt[4:0]); rest of group id is in R0.2. 3: cat(2'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0]). 4: cat(12'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0]). 5: cat(22'b0, OuterGlobalCnt[9:0]).
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
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: MBZ
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 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. Format = U4 Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two
R0.2		
	31	**Barrier ID MSB.** This field is bit[4] of the BarrierID; for full 5-bit barrier ID it should be combined with Barrier ID[3:0]. Format: U1
	30	**Media Group ID.** This dispatch originated in a Media_Walker or Media_Object_GrpId, and so the Thread Group ID is formed from either the loop counters (for Media_Walker) or a supplied group ID (for Media_Object_Grpid).
	29	**Barrier Enable.** This field indicates that a barrier has been allocated for this kernel.
	28	**SLM Enable.** This field indicates that Shared Local Memory has been allocated for this kernel.
	27:24	**BarrierID.** This field indicates the barrier that this kernel is associated with. There is a fifth bit for the BarrierID, in bit 31. Format: U4
	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

DWord	Bits	Description
		descriptors. Format = U6
	3:0	**Scoreboard Color.** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
R0.1	31:27	Reserved: MBZ
	26:16	**Scoreboard Y.** This field provides the Y term of the scoreboard value of the current thread. Format = U11
	15:11	Reserved: MBZ
	10:0	**Scoreboard X.** This field provides the X term of the scoreboard value of the current thread. Format = U11
R0.0	31:30	Reserved: MBZ
	29:24	**SLM ID.** This field indicates the index to the SLM starting offset associated with this kernel. Format: U6
	23:16	**Scoreboard Mask.** Each bit indicates the corresponding dependency that the scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. **Bit n (for n = 0..7)**: Scoreboard n is dependent, where bit 23 maps to n = 0. Format = TRUE/FALSE
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

Thread Spawn Message

The thread spawn message is issued to the TS unit by a thread running on an EU. This message contains only one 8-DWord register. The thread spawn message may be used to:

- Spawn a child thread.
- Spawn a root thread (start dispatching a synchronized root thread).
- Dereference an URB handle.
- Indicate a thread termination, dereference other TS managed resource and may or may not dereference URB handle.
- Release a PRT_Fence.

To end a root thread, the end of thread message must be targeted at the thread spawner. In this case, the root thread sends a message with a "dereference resource" in the Opcode field. The thread spawner does *not* snoop the messages sideband to determine when a root thread has ended. Thread Spawner does not track when a child thread terminates, to be consistent a child thread should also terminate with a "dereference resource" message to the Thread Spawner. Software must set the Requester Type (root or child thread) field correctly.

TS dispatches one synchronized root thread upon receiving a 'spawn root thread' message (from a synchronization thread). The synchronizing thread must send the number of 'spawn root thread' message exactly the same as the subsequent 'synchronized root thread'. No more, no less. Otherwise, hardware behavior is undefined.

URB Handle Offset field in this message (in M0.4) has 10 bits, allowing addressing of a large URB space. However, when a parent thread writes into the URB, it subjects to the maximum URB offset limitation of the URB write message, which is only 6 bits (see Unified Return Buffer Chapter for details). In this case, the parent thread may have to modify the URB Return Handle 0 field of the URB write message to subdivide the large URB space that the thread manages.

Only a persistent root thread can use this message to dispatch a root thread if preemption exceptions are possible. The root thread requested by this message is not guaranteed to dispatch, and the persistent root thread must handle the case where it does not dispatch. When a context switch interrupt is recognized by the persistent root thread, all other root threads that had been dispatched have completed and no more will be dispatched.

Child threads requested by this message are guaranteed to dispatch in all cases, so long as the persistent root thread does not also dispatch synchronized root threads. A child thread does not dispatch if it is behind a synchronized root thread that is not dispatched due to a preemption exception.

In addition to monitor 'end of thread message' targeted to Thread Spawner, Thread Spawner also monitors the message targeting to Message Gateway for EOT signal. Therefore, a child thread, who doesn't hold any hardware resource (URB handle or scratch memory) that Thread Spawner manages, can terminate with a Gateway message with EOT on. The reason of this new TS feature is to avoid a possible risk condition as described below.

In a system running child threads, a parent thread is monitoring the status of the child threads by communications through Message Gateway. When a child thread is about to terminate, it sends a message to the parent through Message Gateway and then sends a second message of EOT (end of thread) to TS.

There is a latency between sending a message to parent thread and the EOT to TS due to message bus arbitration. The parent thread may acknowledge the GW message and issue a new child dispatch before the EOT was processed; basically threads are issued faster than retired.

Because the messages for new child dispatch and EOT go to the same queue in TS, if the queue gets full, EOTs will get blocked. In the case when all the EUs/Threads are full, this will create a system deadlock: no EOTs can be acknowledged by TS (to free up EU resource) and no child threads can be dispatched (to free up TS queue to receive EOT message).

Message Descriptor

The following table shows the lower 20 bits of the message descriptor within the SEND instruction for a thread spawn message.

Thread Spawn Message Descriptor

Message Payload

DWord	Bits	Description
M0.5	31:10	Ignored.
	9:0	**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 a root thread upon thread completion. This field is valid only if the **Opcode** is “dereference resource”, and is ignored by hardware otherwise.
M0.4	31:16	Ignored.
	15:10	**Dispatch URB Length.** Indicates the number of 8-DWord URB entries contained in the Dispatch URB Handle that will be dispatched. When spawning a child thread, the URB handle contains most of the child thread's payload including the R0 header. When spawning a root thread, the URB handle contains the message passed from the requesting thread to the spawned “peer” root thread. The number of GRF registers that are initialized at the start of the spawned child thread is the sum of this field and the number of URB constants if present. The number of GRF registers that are initialized at the start of a spawned root thread is the sum of this field, the number of URB constants if present, and the URB handle received from VFE. This field is ignored if the Opcode is “dereference resource”. A Length of 0 can be used while spawning child threads to indicate that there is no payload beyond the required R0 header. A Length of 0 while spawning a root thread indicates that there is no payload at all from the parent thread. A spawned root has R0 supplied by the Media_Object command indirect/inline data. Format = U6 Range = [0,63] for child threads.
	9:0	**URB Handle Offset.** Specifies the 8-DWord URB entry offset into the URB handle that determines where the associated dispatch payload will be retrieved from when the spawned child or root thread is dispatched. This field is ignored if the **Opcode** is “dereference resource”. Format = U10 Range = [0,1023]
M0.3	31:0	Ignored.
M0.2	31	**Barrier ID MSB.** This field is bit[4] of the BarrierID; for full 5-bit barrier ID it should be combined with Barrier ID[3:0]. Format: U1
	30	Ignored.
	29	**Barrier Enable.** This field indicates that a barrier has been allocated for this kernel.
	28	**SLM Enable.** This field indicates that Shared Local Memory has been allocated for this kernel.
	27:24	**BarrierID.** This field indicates which one of the 16 Barriers this kernel is associated with. Format: U4
	23:16	Ignored.
	15:10	Ignored.

DWord	Bits	Description
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX). This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
M0.1	31:0	Ignored.
M0.0	31:24	**SLM ID.** This field encodes the SLM bank ID and SLM offset in the bank, for the shared local memory for the thread group.
	23:16	Reserved: MBZ
	15:0	**Dispatch URB Handle.** If **Opcode** (and **Requester Type**) is "spawn a child thread": Specifies the URB handle for the child thread. If **Opcode** (and **Requester Type**) is "spawn a root thread": Specifies the URB handle containing message (e.g. requester's gateway information) from the requesting thread to the spawned root thread. If **Opcode** is "dereference resource": This field is required on end of thread messages if the **Children Present** bit is set, as the handle must be dereferenced, otherwise this field is ignored.

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. 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

[Structure] Thread EOT Message Descriptor

Message Payload

DWord	Bits	Description
M0.5	31:8	Ignored.
	7:0	**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 a root thread upon thread completion.
M0.4	31:0	Ignored.
M0.3	31:0	Ignored.
M0.2	31	**Barrier ID MSB.** This field is bit[4] of the BarrierID; for full 5-bit barrier ID it should be combined with Barrier ID[3:0]. Format: U1
	30	Reserved: MBZ
	29	**Barrier Enable.** This field indicates that a barrier was allocated for this kernel.

DWord	Bits	Description
	28	**SLM Enable.** This field indicates that Shared Local Memory was allocated for this kernel.
	27:24	**BarrierID.** This field indicates which one of the 16 Barriers this kernel is associated with. There is a fifth bit for the BarrierID, in bit 31. Format: U4
	23:0	Ignored.
M0.1	31:0	Ignored.
M0.0	31:24	**SLM ID.** This field encodes the SLM bank ID and SLM offset in the bank, for the shared local memory for the thread group.
	23:0	Ignored.

L3 Cache and URB

This section describes the GFX L3 Cache, which is a large storage (introduced in IVB) that backs up various L2/L1 caches on many clients. It provides a simple way based partitioning option for each of a cluster of clients to get a dedicated chunk of the cache. It also acts as a GFX URB and can be configured as highly banked memory for EUs/ROWs.

L3 Cache and URB

This is the BSpec volume L3/URB/SLM.

LBCF Registers

Register
L3SQCREG1 - L3 SQCRegisters1
L3SQCregisters2
L3SQCregisters3
L3ControlRegister1
L3SLMRegister
L3SQCregister4
SCRATCH1
LTCDconfigregister
LBSconfigbits
L3BankStatus
LBCFconfigsavemsg

LNCF Registers

Register
ArbiterControlRegister
SCRATCHforLNCFunit
Eventselectionandbasecounters
EventSelectionandBaseCounters1
LPFCcontrolregister
LNCFconfigsavemsg
LNCFMOCSRegister0
LNCFMOCSRegister1
LNCFMOCSRegister2
LNCFMOCSRegister3
LNCFMOCSRegister4
LNCFMOCSRegister5
LNCFMOCSRegister6
LNCFMOCSRegister7
LNCFMOCSRegister8
LNCFMOCSRegister9
LNCFMOCSRegister10
LNCFMOCSRegister11
LNCFMOCSRegister12
LNCFMOCSRegister13
LNCFMOCSRegister14
LNCFMOCSRegister15
LNCFMOCSRegister16
LNCFMOCSRegister17
LNCFMOCSRegister18
LNCFMOCSRegister19
LNCFMOCSRegister20
LNCFMOCSRegister21
LNCFMOCSRegister22
LNCFMOCSRegister23
LNCFMOCSRegister24
LNCFMOCSRegister25
LNCFMOCSRegister26
LNCFMOCSRegister27
LNCFMOCSRegister28
LNCFMOCSRegister29
LNCFMOCSRegister30
LNCFMOCSRegister31

LPFC Registers

Register
PMflushandslmsaverestore
GlobalInvalidationRegister
PowercontextSaveRegisterforLPFC
FirstBufferSizeandStart
SecondBufferSize
ErrorReportingRegister
FramecountandDrawcallnumber
SaveTimer
Masterstarttimer

Overview

The changes to L3 for SKL are minor. The overall organization of the L3 and theory of operation have not changed. It retains all the performance enhancements from previous generations with some additional improvements.

To provide the bandwidth needed L3 is still organized into independent banks which can be accessed concurrently. Clocking remains 2X in the arrays to balance bandwidth with incoming requests. The L3$ and URB data for all L3 banks are shared as a contiguous memory space for all products regardless of how many banks or slices.

- Each Bank 192KB in size.
- Each logical bank consists of:
    - Data Array
    - Tag Array
    - LRU Array (implements a Pseudo Least Recently Used algorithm)
    - State Array
    - SuperQ Buffer
    - Atomic Processing Units
- The rest of the support logic around L3 consists of:
    - SuperQ (main scheduler).
    - Ingress/Egress queues to L3/SQ (L3 arbiter).
    - CAM structures to maintain coherency.
    - Crossbars for data routing.
- Use of 2x/1x clocking.
- L3 can operate concurrently in GFX and IA coherent domain.
- A portion of L3 can be allocated as highly banked memory and/or unified buffer (URB).

L3 Bank Configuration

Each L3 bank is identical as described below. In products where there is only a single L3 bank supported, the bank will be different to support a larger data array. The Multi-bank describes the L3 Bank configuration for all other products.

Multi-Bank: 192KB data-array and 96 logical ways:

- Up to 64 ways, up to 128KB, tagged for L3$, remaining is treated as memory.
- 64KB or 96KB allocated for URB (does not use tag).
- Shared local memory (SLM) capable (64KB)
    - Uses Highly-banked memory to allow up to 16 32-bit accesses in parallel within a 64KB space per sub-slice.
    - Highly-banked memory formed from 16x4KB arrays

All Bank Implementations:

- 64B Cacheline.
- Interface 64B to SQDB for the fill/write path, 64B Read/Evict path to SQDB. Additional 64B read and 64B write capability for SLM.
- Data protection via ECC.
- TAG/LRU/STATE (using gen-ram via RLS flows):
    - 39/48-bit addressing support in TAG.
    - 6-bit state (2-bits of MESI, 2-bits of Surface type, 1 bit Phys/Virtual, 1-bit Global/Local).
    - Intel pseudo-LRU implementation for selecting the line to be replaced or 1b LRU (added for Gen9).

Bandwidth and Throughput Capability

Programming Note	
Context:	Bandwidth and Throughput Capability

Note: The table below shows the throughput capability for the L3/URB and SLM per bank. Note that L3$ and URB share the same pipelines, so the throughput for L3$ and URB is shared as well. SLM has dedicated pipelining and therefore has its own throughput capability. Total system throughput is derived by multiplying the throughput numbers below by the total number of L3 Banks supported by a product. There is an additional limitation on throughput based the source of the transactions. A single client (e.g. an HDC or sub-slice) can only issue one 64-byte read or write per clock.

Memory Type	Read Throughput (Bytes/Clock)	Write Throughput (Bytes/Clock)	Atomic Throughput (32b Ops/Clock)
L3$ or URB	64	64	16
SLM	64	64	16

L3 Blocks Overview

L3 is formed via some number of logical banks that are identical to each other. The major blocks in each logical bank are:

- L3 Cache Arrays & Controller
- Super Q and related data buffer
- Ingress queues and related CAMs with arbitration
- Atomics Block/SLM pipeline & crossbar for data routing

Our vision is to build a compute scalable cache where with each additional compute both the size and bandwidth are scaled while maintaining the functional single cache concept. Each added bank becomes an additional cache rather than an independent content. The concept is to be able to keep a single copy of a line and service all requesters via distributing their accesses over many physical caches.

Size of L3 Bank and Allocations

Multi-Bank Allocation Options

The table below shows the size of the L3 bank and the ranges of allocation for L3$, URB, and SLM. Below is a table of all validated configurations supported.

No Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
0	0	96	96	0	0	192
1	0	96	0	32	64	192
2	0	64	0	128	0	192
3	0	64	0	0	128	192
4	0	64	128	0	0	192
Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
1	64	32	96	0	0	192
2	64	32	0	32	64	192
3	64	32	0	64	32	192
4	64	16	112	0	0	192

Essentially, the L3 block can either support SLM or not. When SLM is supported, each slice provides 64KB of its 192KB data array for SLM. When SLM is not supported, the 64KB is allocated to URB or split equally between URB and L3$.

The number of L3 Banks will vary for different products and SKUs. The number of banks supported for each product is defined in the Configurations section of the BSPEC. The total amount of L3$, URB, and SLM supported by a product can be calculated by multiplying the number of banks by the values in the above tables.

L3 Cache Theory of Operation

L3/URB operation is required for various clients to access L3 as their back-up cache or memory space. The following clients are listed as L3/URB clients:

L3 Clients:

- Data Cluster (i.e. spill/fills, load/stores, global memory accesses, ...) (read/write)
- Sampler (L2$ - MT) (read)
- IME (motion estimation) (read)
- I$ (Instruction Cache) (read)
- State Arbiter (L3 is state cache replacement) (read)
- Constant cache (read)
- Tesselator (always un-cacheable for SW tessellation)

URB Clients:

- TD-L (read client) – Local Thread Dispatcher
- SBE (read client) – SF Backend
- SOL (read client) – Stream Out
- CL (read client) – Clipper
- GS (read client) – Geometry Shader
- TE (read Client) – Tessalator
- DC (read/write client) – Data Cluster
- VF/(VFE/CS) (write client) – VF acts on behalf of all

L3 vs. URB accesses are separated with a simple field on the request field, for most clients this is only one direction with the exception of data port where both could be addressed L3 or URB. The destination field is part of the request and could be set to re-direct the request to:

- L3 cache
- URB
- State Arbiter
- SLM (highly banked memory)

L3 access/cacheability is determined via request field as well, such parameter will be part of the surface state or base address programming of L3 clients and will be communicated to L3 Cluster along with the request packet.

SLM is accessed in parallel with L3$/URB accesses. L3$/URB share the normal array, and must be serialized. SLM accesses have a dedicate highly-banked array which has lower latency and is not serialized with L3$/URB accesses.

L3 Cache Theory of Operation

L3/URB operation is required for various clients to access L3 as their back-up cache or memory space. The following clients are listed as L3/URB clients:

L3 Clients
Data Cluster (i.e. spill/fills, load/stores, global memory accesses, ...(read/write)
Sampler (L2$ - MT) (read)
IME (motion estimation) (read)
I$ (Instruction Cache) (read)
State Arbiter (L3 is state cache replacement) (read)
Constant cache (read)
Tesselator (always un-cacheable for SW tessellation)

URB Clients:

- TD-L (read client) – Local Thread Dispatcher
- SBE (read client) – SF Backend
- SOL (read client) – Stream Out
- CL (read client) – Clipper
- GS (read client) – Geometry Shader
- TE (read Client) – Tessalator
- DC (read/write client) – Data Cluster
- VF/(VFE/CS) (write client) – VF acts on behalf of all

L3 vs. URB accesses are separated with a simple field on the request field, for most clients this is only one direction with the exception of data port where both could be addressed L3 or URB. The destination field is part of the request and could be set to re-direct the request to:

- L3 cache
- URB
- State Arbiter
- SLM (highly banked memory)

L3 access/cacheability is determined via request field as well, such parameter will be part of the surface state or base address programming of L3 clients and will be communicated to L3 Cluster along with the request packet.

SLM is accessed in parallel with L3$/URB accesses. L3$/URB share the normal array, and must be serialized. SLM accesses have a dedicate highly-banked array which has lower latency and is not serialized with L3$/URB accesses.

Access Policy

L3 partitioning has been reduced to remove some of the complexity. The read-only space clients are merged to have combined space which is allocated over the entire L3 cache and shared with read/write space. The Global vs Local distinction is removed with the coherency being updated.

The destination field should reflect:

- L3 R/W data
- L3 RO data
- URB
- SLM
- State ARB (SARB)

Arbitration Policies

There are three types of L3 transactions which are arbitrated between for Tag access: Snoops, Fills, and Reads (writes are done as full cache-line writes and therefore are treated like fills). Snoops requests arrive from a dedicated queue (LBS), and Fill requests arrive from the LSQC and both are deposited in dedicated ingress FIFOs. Read requests arrive from the LSQC but are sent to two parallel ingress FIFOs. So, at any given time one snoop, one fill, and two reads can be requesting tag access. Once a request is submitted from the LSQC or LBS to the Tag Ingress FIFOs, they cannot be re-ordered. Only the head of the FIFOs is arbitrated against. This is sub-optimal, but much simpler than attempting to re-order within the ingress FIFOs.

The transactions are arbitrated in the 1x clk domain but are submitted into the 2x clk domain (the tag, state, LRU, and data arrays operate in the 2x clk domain). So, up to two commands can be submitted to the tag pipeline in a single 1x clk.

Tag Request Arbitration

The arbitration for tag access is based on cycle type and set address:

- Snoops have highest priority. Whenever a snoop is presented, it takes precedence over Fills and Reads. Only one snoop can be done at a time (per 1x clk).
- Fills have priority over Reads. Only one fill can be done at a time.
- Reads/Atomics take any empty arbitration slots. Since two commands can be submitted for each 1x clock, that means up to two reads can be submitted at a time (per 1x clk).
- No two transactions to the same set address are allowed. In this case one of the ports will be held idle to insert a bubble in the pipeline. This set address dependency is caused by resource conflicts

in both the Tag and LRU arrays. For Gen9, special buffers will be added to contain results from the last one or two aromic operations, to allow for back-to-back atomics to the same CL. This eliminates the need for data array accesses, which eliminates a bubble in the data pipelines.

- Only L3 accesses request Tag access. URB and SLM requests go directly to the data array pipeline.

Data Request Arbitration

The data-array is accessed by the LSQD whenever a read operation (for data read or eviction) or fill is required. The LSQD receives commands from the LSQC based on the tag response (in the case of L3 accesses) or for URB accesses without any need for tag lookup.

SLM data accesses come from a separate pipeline. However, SLM accesses are always to the dedicated highly-banked memory and do NOT require arbitration against L3/URB requests. In the case where SLM is not enabled, then the highly-banked memory is allocated to URB. In this case, data returns from the highly-banked memory will need to be muxed with the data from the larger data array.

L3 (from the Tag pipeline), URB, and Snoop-Evictions must be arbitrated for access to the data array. The arbitration is a fixed priority:

with the Snoops having highest priority and URB having next highest priority.

Back-to-back reads and back-to-back writes to the data array will always be prevented through arbitration. A 2-clock (2x clk) bubble will be inserted to allow the data array to recover and precharge as required.

The highly banked memory still has the same back-to-back read or back-to-back write restriction.

L3 Ordering Restrictions

The LSQC will enfore specific ordering requirements on the accesses to the L3/URB, but will still allow out-of-order accesses where possible.

Basic Ordering Requirements

Requirements:

1. Ordering of transactions primary purpose is to ensure coherency and causality for software accesses to memory. For example, if a thread writes to a memory address, it expects that any subsequent read issued by the same thread to the same address should read back what it wrote. Conversely, if the read is before the write, it should not read back what it wrote.
2. Transactions which have hardware constraints must be ordered to avoid conflicts. For example, two FP-Atomic Add operations must be ordered to avoid conflict on a single FP Adder.

With these two requirements in mind, the order requirements are defined.

Rules:

1. All reads/fills and atomics to the same cacheline must be ordered and serialized such the oldest is completed first and the youngest is completed last. This means that an older read/fill in the SQ must be completed prior to a younger read/fill being started. This meets requirement #1 above. Strictly speaking two or more reads to the same cacheline need not be ordered, but there's no particular advantage to re-ordering these.
2. All reads/fills/atomics to a cacheline can proceed even if an eviction is pending for that cacheline. In practice, holding a read/fill until an eviction is completed is not necessary because the L3 allocation policy (see next section) is such that allocation of a new line into a set is not done until the Fill arrives. So, any read/fill to the old (to be evicted) line can proceed normally. Once the Fill does arrive, the eviction of the line and update of the tag will make any subsequent read/fill a miss and cause that transaction to reallocate through the normal flows.
3. Two or more evictions of the same cacheline must be ordered with the oldest going first and the youngest going last. Rule #2 above creates a funny boundary case where you can have two evictions to the same cacheline in the LQSC at the same time. This is possible in a scenario where you have a dirty fill which causes an eviction to a cacheline in a set, followed by a long series of dirty evictions to the same set. Eventually, if the Fill for your first line is taking a very long time to return, it is possible for a 2nd eviction to that same line to occur. For coherent lines, this cannot occur because any coherent cycle would issue a self-snoop which would cause the eviction to be pushed prior to any subsequent eviction (see Rule #5).
4. All FP-atomics to any cacheline are ordered and serialized. This is done to break the conflict on use of the Floating-Point ALU hardware. This meets requirement #2 above.
5. All snoops must be ordered with respect to evictions to the same line. Snoops have their own queue, but they also check the contents of the SQ for evictions to the same line. In other words, snoops should push older evictions to the same cacheline.

This is summarized below. Empty cells denote "no ordering/serialization required". A "Yes" indicates an ordering/serialization between the transactions on the row and column. The "Same CL" indicates that the ordering/serialization is only for transactions to the same cache-line.

Transaction Ordering Rules for Transactions to the L3/URB

Transaction	Read/Fill	Eviction	Int. Atomic	Snoop	FP-Atomic
Read/Fill	Yes, same CL		Yes, same CL		Yes, same CL
Eviction		Yes, same CL			
Int-Atomic	Yes, same CL		Yes, same CL		Yes, same CL
Snoop		Yes, same CL			
FP-Atomic	Yes, same CL		Yes, Same CL		Yes

Note, that the ordering/serialization described above does not limit accesses to the TAG. When two accesses arrive to the same cache-line in the SuperQ, we still form the parent-child link to allow for serialization of the older before the younger, however the younger can still proceed with TAG look-up as soon as older sends its TAG look up to L3 without waiting for the result of the older. If older is a HIT, the child will get a HIT as well hence no issue. If older is a MISS, the younger will get a MISS as well. In this case the older is the only one launching a read (FILL) to memory. Younger will be stalled until older completes the FILL. Child will replay the TAG look up after FILL operation. If there are more CHILD accesses they will immediately play once the second CHILD is sent to TAG pipeline. This optimization allows for a reduction in overall latency for pending requests by allowing tag look-ups for HITs to be done ahead of time.

Allocation Policy

During the MISS time of L3 look-up there is no pre-allocation. Only after the miss data is returned from LLC/Memory, the fill will be sent to data array (it is guaranteed not a match due to single address processing) and allocation will take place during the Fill time.

If the allocated entry is not modified, than FILL will overwrite the location and pipeline moves on. If the allocated entry carries a dirty data, eviction takes place and dirty data will be moved to SQ for writing out to memory. The Tag and state will only be updated once the fill and any eviction is actually being done.

The read for eviction and the fill operation are considered atomic. Only one transaction is issued from the super Q, but it creates a Read followed by a Fill. Victim data is returned to the SQDB.

Allocation on Fill eliminates many boundary cases via regulating the entry invalidation at the last phase of the data servicing. However, it also creates boundary cases which are handled in the LSQC.

High Level Algorithm Descriptions

The pLRU algorithm uses a binary decision tree with (N-1) nodes. Each node is controlled by a single bit. When a way is filled, all the nodes in the tree to reach that way are flipped to point to another way which is essentially as far away as you can get. The new cache-line is guaranteed to be retained for N-2 additional fills to the set. The LRU bits are only updated on Fills.

An N-way 1b LRU has an N-bit vector for each set. The vector is initialized to all 0's. As each Fill request arrives, the first bit in the vector with a 0 is selected, that way is replaced, and the bit is flipped. Each subsequent fill will search for the first 0 and replace that line and flip it's bit. Eventually, when all N bits are 1 and there are no candidates, all bits are cleared and the first way is selected.

Both implementations use approximately the same amount of logic with the N-1 bits per bits. One uses a binary tree to decode and selects bits, whereas the other has to handle read-hits.

High Level Algorithm Descriptions

The pLRU algorithm uses a binary decision tree with (N-1) nodes. Each node is controlled by a single bit. When a way is filled, all the nodes in the tree to reach that way are flipped to point to another way which

is essentially as far away as you can get. The new cache-line is guaranteed to be retained for N-2 additional fills to the set. The LRU bits are only updated on Fills.

An N-way 1b LRU has an N-bit vector for each set. The vector is initialized to all 0's. As each Fill request arrives, the first bit in the vector with a 0 is selected, that way is replaced, and the bit is flipped. Each subsequent fill will search for the first 0 and replace that line and flip it's bit. Eventually, when all N bits are 1 and there are no candidates, all bits are cleared and the first way is selected.

Both implementations use approximately the same amount of logic with the N-1 bits per bits. One uses a binary tree to decode and selects bits, whereas the other has to handle read-hits.

Memory Object Control State on Cacheability

Memory Object Control State information has been consolidated under the memory view section of the BSpec.

Overview

The changes to L3 for SKL are minor. The overall organization of the L3 and theory of operation have not changed. It retains all the performance enhancements from previous generations with some additional improvements.

To provide the bandwidth needed L3 is still organized into independent banks which can be accessed concurrently. Clocking remains 2X in the arrays to balance bandwidth with incoming requests. The L3$ and URB data for all L3 banks are shared as a contiguous memory space for all products regardless of how many banks or slices.

- Each Bank 192KB in size.
- Each logical bank consists of:
    - Data Array
    - Tag Array
    - LRU Array (implements a Pseudo Least Recently Used algorithm)
    - State Array
    - SuperQ Buffer
    - Atomic Processing Units
- The rest of the support logic around L3 consists of:
    - SuperQ (main scheduler).
    - Ingress/Egress queues to L3/SQ (L3 arbiter).
    - CAM structures to maintain coherency.
    - Crossbars for data routing.
- Use of 2x/1x clocking.
- L3 can operate concurrently in GFX and IA coherent domain.
- A portion of L3 can be allocated as highly banked memory and/or unified buffer (URB).

Atomics

An atomic operation may involve both reading from and then writing to a memory location. Atomic operations apply only to either u# (Unordered Access Views) or g# (Thread Group Shared Memory). It is guaranteed that when a thread issues an atomic operation on a memory address, no write to the same address from outside the current atomic operation by any thread can occur between the atomic read and write.

If multiple atomic operations from different threads target the same address, the operations are serialized in an undefined order. This serialization occurs due to L3 serialization rules to the same address.

Atomic operations do not imply a memory or thread fence. If the program author/compiler does not make appropriate use of fences, it is not guaranteed that all threads see the result of any given memory operation at the same time, or in any particular order with respect to updates to other memory addresses. However atomic operations are always stated on a global level (except on shared local memory), when atomic is operation is complete final result is always visible to all thread groups.

Each generation since Gen7 has increased the capability and performance of atomic operations.

Gen8 atomics introduced these features:

- Double size operands where 8B atomic operations are introduced for 64-bit data types. There is also an addition 16B "atomic_cmp/wr16B" to the table. All these new operands apply to global memory only.
- Floats for 4B/8B accesses, only floating point adder is used.
    - (*note: if L3 atomics are disabled, floating point atomics can be used*)
- Move global atomic ops to L3 (keep the GTI support for non-coherent L3 mode) and share same atomic OPs as SLM.
    - GT3 can process up to 192 IA coherent atomics per cycle.

Gen 9 adds these features:

- Hardware improves the performance of atomics to the same cacheline. Specifically, back-to-back atomics to the same memory location are done without the need to re-fetch the result of the previous atomic from L3. This should increase bandwidth for atomics to the same location significantly over Gen8/BDW.
- Single-Precision FP Min/Max and Compare/Exchange Instructions are added.

Gen 9 atomics hitting an overflow condition for memory buffer may corrupt other atomics.

Programming Note	
Context:	Atomics
API Specification: The API specification says that atomic operations on Thread Group Shared Memory are atomic with respect to other atomic operations, as well as operations that only perform reads (loads). However atomic operations on Thread Group Shared Memory are *not*atomic with respect to operations that perform only writes	

Programming Note	
Context:	Atomics
(stores) to memory. Mixing atomics and stores on the same Thread Group Shared Memory address without thread synchronization and memory fencing between them produces undefined results at the address involved. This restriction arises because some implementations of loads and stores do not honor the locking semantics for implementing atomics. It turns out this has no impact on loads, since they are guaranteed to retrieve a value either before or after an atomic (they will not retrieve partially updated values, given they are all defined at 32-bit quanta). However store operations could find their way into the middle of an atomic operation and thus have their effect possibly lost.	

For Gen9 we do NOT allow stores to break an atomic.

In L3 or SLM, the atomic operation leads to a read-modify-write operation on the destination location with the option of returning value back to requester. The table below is defined as a list of atomic operations needed:

Atomic Operation	Opcode	Description	New Destination Value	Applicable	Return Value (Optional)
Atomic_AND	0000_0001	Single component 32-bit bitwise AND of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	“old_dst” AND “src0”	global/SLM	old_dst
Atomic_OR	0000_0010	Single component 32-bit bitwise OR of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	“old_dst” OR “src0”	global/SLM	old_dst
Atomic_XOR	0000_0011	Single component 32-bit bitwise XOR of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	“old_dst” XOR “src0”	global/SLM	old_dst
Atomic_MOVE	0000_0100	Replace the dst with src0.	“src0”	global/SLM	old_dst
Atomic_INC	0000_0101	Single component 32-bit integer increment of dst back into dst.	“old_dst + 1”	global/SLM	old_dst
Atomic_DEC	0000_0110	Single component 32-bit integer decrement of dst back into dst.	“old_dst - 1”	global/SLM	old_dst
Atomic_ADD	0000_0111	Single component 32-bit integer add of operand src0 into dst at 32-bit per component address dstAddress, performed atomically. Insensitive to sign.	“old_dst + src0”	global/SLM	old_dst
Atomic_SUB	0000_1000	Single component 32-bit integer subtraction of operand src0 from dst at 32-bit per component address	“old_dst - src0”	global/SLM	old_dst

Atomic Operation	Opcode	Description	New Destination Value	Applicable	Return Value (Optional)
		dstAddress, performed atomically. Insensitive to sign.			
Atomic_RSUB	0000_1001	Single component 32-bit integer subtraction of operand dst from src0 into dst at 32-bit per component address dstAddress, performed atomically. Insensitive to sign.	“src0 - old_dst”	global/SLM	old_dst
Atomic_IMAX	0000_1010	Single component 32-bit signed MAX of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	IMAX(old_dst, src0)	global/SLM	old_dst
Atomic_IMIN	0000_1011	Single component 32-bit signed MIN of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	IMIN(old_dst, src0)	global/SLM	old_dst
Atomic_UMAX	0000_1100	Single component 32-bit unsigned MAX of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	UMAX(old_dst, src0)	global/SLM	old_dst
Atomic_UMIN	0000_1101	Single component 32-bit unsigned MIN of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.	UMIN(old_dst, src0)	global/SLM	old_dst
Atomic_ CMP/WR	0000_1110	Single component 32-bit value compare of operand src0 with dst at 32-bit per component address dstAddress.	(src0 == old_dst)?	global/SLM	old_dst
		If the compared values are identical, the single-component 32-bit value in src1 is written to destination memory, else the destination is not changed.	src1:		
		The entire compare+write operation is performed atomically.	old_dst		
Atomic_ PREDEC	0000_1111	Single component 32-bit integer decrement of dst back into dst.	“old_dst - 1”	global/SLM	new_dst
Atomic_AND8B	0010_0001	Single component 64-bit bitwise AND of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	“old_dst8B” AND “src08B”	global	old_dst8B

Atomic Operation	Opcode	Description	New Destination Value	Applicable	Return Value (Optional)
Atomic_OR8B	0010_0010	Single component 64-bit bitwise OR of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	“old_dst8B” OR “src08B”	global	old_dst8B
Atomic_XOR8B	0010_0011	Single component 64-bit bitwise XOR of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	“old_dst8B” XOR “src08B”	global	old_dst8B
Atomic_ MOVE8B	0010_0100	Replacement of the dst with src0.	“src08B”	global	old_dst8B
Atomic_INC8B	0010_0101	Single component 64-bit integer increment of dst back into dst	“old_dst8B + 1”	global	old_dst8B
Atomic_DEC8B	0010_0110	Single component 64-bit integer decrement of dst back into dst	“old_dst8B - 1”	global	old_dst8B
Atomic_ADD8B	0010_0111	Single component 64-bit integer add of operand src0 into dst at 64-bit per component address dstAddress, performed atomically. Insensitive to sign.	“old_dst8B + src08B”	global	old_dst8B
Atomic_SUB8B	0010_1000	Single component 64-bit integer subtraction of operand src0 from dst at 64-bit per component address dstAddress, performed atomically. Insensitive to sign.	“old_dst8B - src08B”	global	old_dst8B
Atomic_ RSUB8B	0010_1001	Single component 64-bit integer subtraction of operand dst from src0 into dst at 64-bit per component address dstAddress, performed atomically. Insensitive to sign.	“src08B - old_dst8B”	global	old_dst8B
Atomic_IMAX8B	0010_1010	Single component 64-bit signed MAX of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	IMAX (old_dst8B, src08B)	global	old_dst8B
Atomic_IMIN8B	0010_1011	Single component 64-bit signed MIN of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	IMIN (old_dst8B, src08B)	global	old_dst8B
Atomic_	0010_1100	Single component 64-bit unsigned MAX of operand src0 into dst at 64-bit per component address dstAddress,	UMAX (old_dst8B, src08B)	global	old_dst8B

Atomic Operation	Opcode	Description	New Destination Value	Applicable	Return Value (Optional)
UMAX8B		performed atomically.			
Atomic_ UMIN8B	0010_1101	Single component 64-bit unsigned MIN of operand src0 into dst at 64-bit per component address dstAddress, performed atomically.	UMIN (old_dst8B, src08B)	global	old_dst8B
Atomic_ CMP/WR8B	0010_1110	Single component 64-bit value compare of operand src0 with dst at 64-bit per component address dstAddress.	(src08B == old_dst8B)?	global	old_dst8B
		If the compared values are identical, the single-component 64-bit value in src1 is written to destination memory, else the destination is not changed.	src18B:		
		The entire compare+write operation is performed atomically.	old_dst8B		
Atomic_ PREDEC8B	0010_1111	Single component 64-bit integer decrement of dst back into dst.	“old_dst8B - 1”	global	new_dst8B
Atomic_ CMP/WR16B	0100_1110	Single component 64-bit value compare of operand src0 with dst at 64-bit per component address dstAddress.	(src0_16B == old_dst16B)?	global	old_dst16B

L3 Coherency

Coherency is one of the crucial topics within L3, there are multiple levels of coherency that are checked and ensured via L3. The list of domains and flows is dependent on the usage models however basic premise is always the same.

The coherency levels:

1. Thread Level Coherency within a Thread Group
2. Thread Group Coherency between multiple domains
3. GPU/IA level coherency

Besides these special domains basic producer/consumer models are followed which are listed as:

1. Fixed function is producing
2. Data Port is producing

All these high level coherency models are investigated and addressed.

Thread Level Coherency

A given thread group is contained within a quarter slice in Gen9, where its writes and reads target the L3 for global memory and SLM for shared local memory. Given the shared local memory view is the same for all quarter slice accesses there is no question about the coherency or data sharing within the thread group. Local syncs are executed up to HDC boundary and not exposed to L3.

Thread Group Coherency

Thread groups can be distributed to multiple quarter slices that are physically far from each other. The coherency between thread groups can only be maintained for their global memory accesses. There are two implications of this coherency depending on the mode we are operating at:

1. Non IA-Coherent L3 mode: This is the same methodology that was introduced on Gen7.5 with the addition of GT4 support. The cross thread group coherency is maintained via Sync Global which is processed by L3 as well as introducing WT mode to be able to update global memory with latest data. This mechanism is supported on Gen9, but not expected to be used according to feedback from software architects.
2. IA-Coherent L3 mode: For the new mode of operation, there is no need to have WT behavior. The data in L3 is already visible to all consumers (i.e. other thread groups of GPU or IA cores). Sync'global has no affect given the data in L3 is already globally visible to all consumers.

GPUIA Level Coherency

In non-coherent L3 mode (i.e. gen7.5 behavior), data sharing between GPU and IA happens via a s/w controlled flow which requires the internal GPU caches to be flushed in order to make the data visible, similarly same caches need to be invalidated when IA produces data. This mode of operation will still be available for gen9. Within the L3, non-coherent accesses use "virtual" addresses and are tagged in the L3 Tag array using the Virtual/Physical bit.

In Gen9, coherent memory is supported where L3 content is visible to IA via snoops. This allows certain streams (i.e. data port) to be GO (Globally Observable) for IA once posted to L3, allowing shorter loop for completions and eliminating the need to do an entire pipeline flush over L3 to get it sync'ed to IA coherent domain. Coherency enhances the general programmability of GPU when cooperating with IA. Within the L3, coherent accesses use a full 48-bit physical address and are tagged using the Virtual/Physical bit in the I3 Tag array.

Coherency Usage Models

This section is to give some examples of usage models and high level handling within L3. They are specific to L3 flows and not meant to represent over coherency usage models on the system level.

Fixed Func Producing (URB)

Fixed functions clients and slice clients consuming is a very common usage model for URB based data sharing. In this mode, FF sends writes to URB and shifts to their pipeline, rest of the pipeline clients including slice clients will read their content from URB.

Process is achieved via URB writes completion from the GO ("Globally observable") point which would be the *slice node* for gen9. Once slice node consumes the write and schedules for BANK the completion is returned back to producer which enables the consumer. At the time the consumers start, the data is already at the point of GO where BANK superQ will keep the ordering.

Note that a node should not be scheduling writes from ingress queues if there are no credits and no progress can be made. The completion from a node is only given once the write from the producer leaves the node ingress queue.

Fixed Func Producing (Push Constants)

Push constants are also similarly processed; in fact the target is the same: URB. However URB accesses are processed with their URB offsets via TAG, their original address is based on virtual memory pointing to buffer location in memory. The delivery mechanism makes the CAMs not possible hence the completions are managed once access is deemed towards SuperQ of the corresponding L3 Bank.

There are additional constraints independent of L3 handling of push constants which are defined as part of the Global Arbitration fabric.

EUs Producing via HDC

Data port is the producer for Global and Shared Local memory types. The shared local memory is specific to a data port, hence L3 does not have to do anything special to maintain coherency but to keep accesses to SLM in-order.

For Global memory coherency is maintained via combination of many mechanisms.

1. Completion tracking: Each HDC tracks their writes towards L3 and wait for them to reach to GO. GO message is given by L3

a. For Non-coherent/virtual addressed Write: Once the ingress queue retires the write in the corresponding node to targeted bank.
b. For Coherent/physical addressed Write: Once the ownership is obtained for the write (i.e. read-for-ownership or invalid-to-modified) which means write is GO with respect to IA cores.

GO information is used within data port to make sure handle releases are gated until the producers updates are globally visible.

2. Thread Level Flush: EU threads have the capability to push globally tagged data from L3 to next level caches. The capability was added as part of Gen7.5 GT4 development for synchronizing cross contents between two cores. It is still supported in Gen9, but it is not used according to software architects. Usage is not recommended.

Invalidation and Flushes

For Gen9, there are two different sources to initiate flushes and invalidations:

- Command Streamer initiated
- EU/thread initiated

Both cases have two modes of operation

- Non-IA coherent L3
- IA coherent L3

In addition there are side flows that back up the various flows and they do require invalidation/flush sequences to be executed for their purpose. Their behavior has no impact between two modes of L3 operation.

- Global Invalidation
- Power Management Invalidation

Node Invalidation Architecture

The idea behind the node architecture is to make each L3 node to be independent and easily scalable; this is done via standardizing the flows and using common interfaces to exchange information between blocks. Invalidations and Flushes are meant to be tuned to similarly to fit into the concept. The final result of invalidation is that all cache-lines related to a surface or type will be invalid in the cache and all modified lines will have been written back to system memory.

Invalidation within a node may be done on a single surface or it may involve the entire cache depending on how the cache is configured and what type of invalidation is being requested. It is up to the node to figure this out and do the right thing. For example, read-only lines will never be written back since they cannot be modified, so they will only be invalidated. Modified read/write lines will be explicitly written back.

Each node performs the invalidation of all candidate lines for its own slice. This is coordinated by the unslice, and invalidation requests are submitted to each slice. Each slice indicates completion back to the unslice.

Each node will have to be notified on all invalidations as soon as possible and their behavior is to make that particular stream un-cacheable even if it did not start the invalidation process already. The process is much simpler when it comes to operations which do not require a feedback back to originator. The command streamer top of the pipe invalidations are under this category. Each node can process these invalidations independently but act on it for turning the particular stream to look uncacheable right away.

For the invalidation and flush processes where feedback is required, the event needs to be synchronized between nodes.

Each node receiving the invalidation or flush will push their ingress FIFOs to make sure their content is also covered as part of the flush event.

Command Streamer Invalidation Flows

Command streamer can do top of the pipe invalidations which are direct connections from command streamer unit to L3 in respective slices, as well as pipeline flush which runs through the data port to respective slice L3 node. Both flows are handled separately.

Any context can have both coherent and non-coherent L3 entries. Hence L3 has no register bit that says it is running in coherent vs non-coherent mode. In the operation time, both modes will co-exist simultaneously where some lines in the cache are following a coherent protocol (i.e. physical address) and remaining lines are following non-coherent protocol (i.e. virtual address).

Non-IA Coherent Flows

This is the traditional flow where the content of L3 needs to be invalidated or flushed similar to gen7.5 flows.

Top of the Pipe Invalidations

For this case, the invalidation causes the cache to drain all FIFOs and pipelines, and the node performs all invalidations. Only after completion in all slices will new transactions be sent to the L3.

The Top of the Pipe invalidation is intended for cases where invalidation and completion need to be coordinated between slices. Therefore, each slice performs invalidation, but will not proceed until all slices have completed the invalidation.

Nodes are responsible to serialize the invalidation and use double buffering for each event.

Pipeline Flush

Pipeline invalidation is much simpler and optimized for performance. In this case there is no need to coordinate between slices and the operation is more like a “sync” point. Transactions arriving after the invalidation request will still be queued in the SQ, but will only be processed after the invalidation of all candidate lines in the cache is complete. The invalidation acts like a fence.

Nodes are responsible to serialize the invalidation and use double buffering for each event.

IA-Coherent Flows

In principle IA-coherent flow is same from node perspective, the only difference is within the banks given the data that we are trying to flush or invalidate is already coherent with IA. Such case eliminates the need to push any bank content explicitly out to LLC.

All that is needed from the bank is to make sure LSQC content is posted into the bank where it is IA visible for a bank to return flush/invalidation completion status to the node.

EUThread Flows

As part of gen9, we continue to support the capability for EUs to be able to perform flush/invalidation events that are not coordinated between other EUs. HDC will relate the request to corresponding L3 and require the entire content of this buffer to be invalidated or flushed to relative coherency domain.

Similar to pipeline flush and top of the pipe invalidations, each node receiving this event will have to communicate with other nodes and make the corresponding traffic un-cacheable before returning the response back to corresponding HDC, allowing progress. The actual invalidation/flush will be deferred and performed once all nodes agree on what needs to be done.

The IA-coherent vs non-IA coherent treatment is same as command streamer flows and still applicable for EU/Thread Flows for invalidation/flush.

Global Invalidation

As part of a mitigation plan to plug holes, a global invalidation flow is introduced where each L3 will get the request from their SARB (or config agent) and again coordinate between the nodes. Once invalidation/flush is performed, completion will be returned back to config agent to clear the flags allowing s/w to observe the end of the global invalidation. Only lines which are tagged as “global” will be flushed in this manner.

Global invalidation will invalidate/flush every line which has its global bit set regardless of whether the line is tagged as virtual or physical (non-coherent or coherent).

Power Management Invalidation

Given we have a IA-coherent mode where a standard pipeline flush does not push the modified lines to outside of GT and we are progressing to deferred invalidations, we need a way to ensure L3 content is cleared and deferred events are complete. This is where PM interaction will have to be introduced.

Corresponding PM will message to L3 config agent to request a flush when needed. Main usage mode is prior to entering RC6. The rest of the treatment in the nodes is the same. Once flush/invalidate event is complete, message(s) will be sent back to PM with the completion status.

L3 Allocation and Programming

L3 Cache allocation is done on a per way basis which should be consistent across all 4 banks (2 banks for GT1). The way allocation between URB and any of the L3 clients can only be changed post pipeline flush where L3 contains no data. This is required for stream based flushes to be dependent on the way allocation of these corresponding streams. S/W should not be removing ways under a particular stream and expect a later pipelined stream flush to target all the corresponding locations. The stream based flush will be performed on the existing way allocation of that stream, there is no history of previous way allocation tracked in the hardware.

L3 Cache has been divided into following client pools:

- Shared Local Memory: When enabled its size is always fixed to 256 KB (128 KB for GT1)
  - 64 KB in the 4th bank per slice is left unused for GT2/3/4 SKUs.
- URB: Local memory space, provides a flexible allocation on per 8 KB granularity
- DC: Data Cluster Data type
- Inst/State: Both instructions and state allocation are combined
- Constants: Pull constants for EUs
- Textures: texture allocation to back-up L2$

In addition to these sub-groups, a collection of groups are generated to bundle multiple clients under the same allocation set:

- All L3 Clients: DC, Inst/State, Constants & Textures
- Read-Only Clients: Inst/State, Constants & Textures

If pLRU is used, the best performance can be attained via assigning a power-of-2 number of ways. This is to ensure pLRU to distribute the ways without hot spotting within that client's group. Even though design provides a flexible (per way basis) programming model for way allocation for each client following table is given for validation and SW programming models. The programming options in the following table represents most likely cases for different operation modes.

L3 cache requires cache control parameter consistency between the surfaces that they have overlapping sections. If multiple buffers are defined to overlap, the cache policy for L3 is required to be the same between them.

L3 Space allocation can only be changed when the GPU pipeline is completely flushed. To guarantee that following two events need to be executed prior to the inline register updates to L3 allocation registers:

1. PIPECONTROL FLUSH, CS Stall set, with HDC Flush set, RO cache invalidation set if required (This flush command ensures the workload is completely drained, Datapipe is completely flushed followed by initiation of RO cache invalidation. Doesn't ensure RO cache invalidation is complete.)

2. PIPECONTROL FLUSH, CS Stall set, with HDC flush. (This flush ensures any prior RO cache invalidation in progress to be complete before processing flush for this command, this avoids RO cache invalidation colliding with following LRI.)

The following settings are given for GT2.

- GT1 has half of the allocation listed below.
- GT3 has 2x the allocation listed below.
- GT4 has 3x the allocation listed below.

(KBytes)Normal Banked - No SLM									
	SLM	URB	Rest	DC	RO(I/S/C/T)	I/S	C	T	Sum
0	0	384	384	0	0	0	0	0	768
1	0	384	0	128	256	0	0	0	768
2	0	256	0	128	384	0	0	0	768
3	0	256	0	0	512	0	0	0	768
4	0	256	512	0	0	0	0	0	768
(KBytes)Shared Local Memory Mode - highly banked M									
	SLM	URB	Rest	DC	RO(I/S/C/T)	I/S	C	T	Sum
1	256	128	384	0	0	0	0	0	768
2	256	128	0	128	256	0	0	0	768
3	256	128	0	256	128	0	0	0	768

Description
URB is limited to 1008KB due to programming restrictions. This is not a restriction of the L3 implementation, but of the FF and other clients. Therefore, in a GT4 implementation it is possible for the programmed allocation of the L3 data array to provide 3*384KB=1152KB for URB, but only 1008KB of this will be used.

Unlike previous generations, logic is simplified via combining RO clients under one pool, given most clients already have their L1 and L2 caches. Bundling them greatly reduces the timing paths and complexity in trying to determine which client belongs to which way.

On Gen9, RW can use the entire 64 ways allocated to L3, and RO can also use the entire 64 ways. In Gen8, RW was restricted to the first 32 ways. So, effectively, the RO and RW memory spaces have been merged into one global space. All ways now have full MESI-state tracking.

Similar to previous generations before changing the configuration of L3, the entire contents needs to be flushed and invalidated. This is to prevent any false hits post configuration update. The invalidation can be achieved either thru command streamer's pipeline flush + top of the pipe invalidations or the global invalidation option within L3. Both options will work for non-coherent option of L3.

For coherent L3 the only option is to follow pipeline invalidation via an L3 global invalidation. That would push all coherent content out of L3 as well allowing a programming update in allocation.

R/W ways allocated (Rest and DC) can not be any size less than 128KB unless they are 0KB.

The locations that are tagged with physical addresses do not need to be flushed between configuration changes.

The following table is added to define the capabilities of programmability in L3.

(KBytes)Normal Banked - No SLM									
SLM	URB	Rest	DC	RO(I/S/C/T)	I/S	C	T	Sum
0	0 to 512 increments of 16KB	0 to 512 increments of 16KB	0	0	0	0	0	768
0	0 to 512 increments of 16KB	0	0 to 512 increments of 16KB	0 to 512 increments of 16KB	0	0	0	768

(KBytes)Shared Local Memory Mode - highly banked M								
SLM	URB	Rest	DC	RO(I/S/C/T)	I/S	C	T	Sum
256	0 to 512 increments of 16KB	0 to 512 increments of 16KB	0	0	0	0	0	768
256	0 to 512 increments of 16KB	0	0 to 512 increments of 16KB	0 to 512 increments of 16KB	0	0	0	768

Programming Notes	
Context:	L3 Allocation and Programming

- Allocating the entire cache to DC (Data space) with 0 KB for reads is not allowed. i.e., when Rest is programmed to 0KB, then RO cannot be 0KB
- However, you can program the entire cache as RO or Rest.
- 64KB in SLM space is left unused for GT2/3/4 SKUs.
- A pipeline flush is required prior to changes in L3 allocation and programming. If the L3 contains coherent lines, it must be ensured that these coherent lines are also flushed through the use of the pipe line flush coherent lines setting in L3SQCREG4 prior to the pipeline flush.
- When changing the L3 mode from SLM to non-SLM or vice versa, a CS stalling pipeline flush is required prior to and following the programming of the SLM Mode Enable bit in the L3CNTLREG.
- In the usage model where data is produced by the data port and consumed via the sampler, the cache

Programming Notes	
Context:	L3 Allocation and Programming
allocation shall always be in the unified mode where "Rest" is allocated a non zero value with no allocations for DC and RO ways. Also, the line replacement mode should be set to 1bLRU only.	

Shared Local Memory

Shared local memory (SLM, also known as highly-banked memory) is a portion of L3 which will be dedicated to EUs as a local memory when enabled.

SLM Behavior and Software Usage

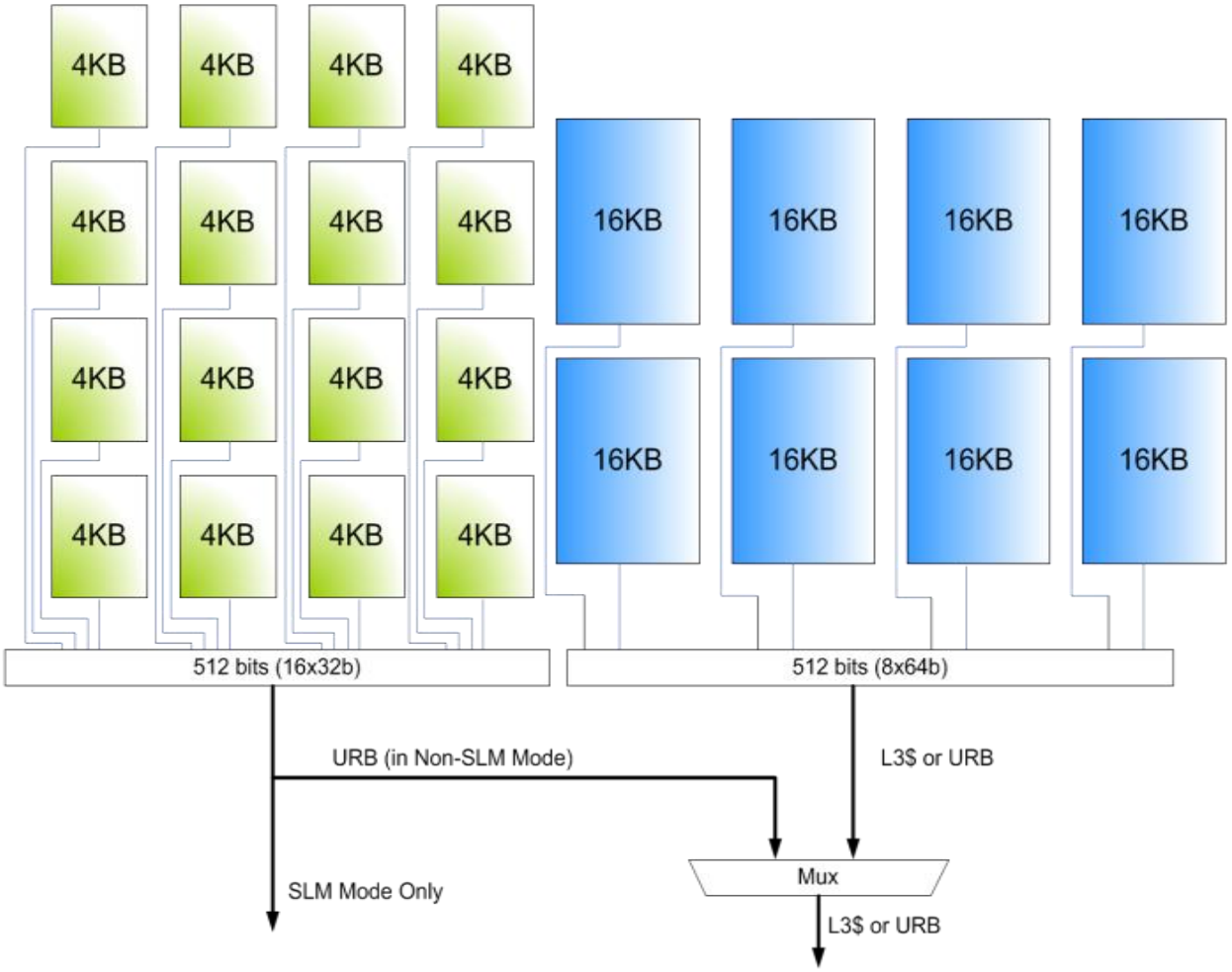
In SKL implementations, the SLM in each physical L3 bank is dedicated to a single sub-slice. All EU's in that sub-slice can access this SLM, but they cannot access the SLM in other L3 banks. SLM is always 64Kbytes in size and is not coherent with system memory. Software is responsible for managing the contents of the SLM.

SLM is enabled statically by the application. When enabled, it uses a fixed 64K block in each L3 Bank. When SLM is enabled, URB will not use this 64K block of memory. URB accesses will automatically be moved to a different location in the L3 bank.

SLM is implemented to allow for highly parallel applications. 16 DWord-accesses (all read or all write) can be done simultaneously to different locations in the 64K space. Atomic operations as described in the Atomic Operation section of this volume can be performed on the output of the SLM and written back in one operation.

SLM Bank Hardware Implementation

The accesses are only possible through data cluster with the destination flag set as SLM. To support a highly banked design, each of the L3 banks are structured to have 16x4KB portion which could be accessed independently per clock. This part of the L3 can support 16 DW size accesses (per SLM) in a given clock cycle. Each bank's SLM is dedicated to a single sub-slice. The diagram below shows the organization of data for a 192KB L3 Bank.

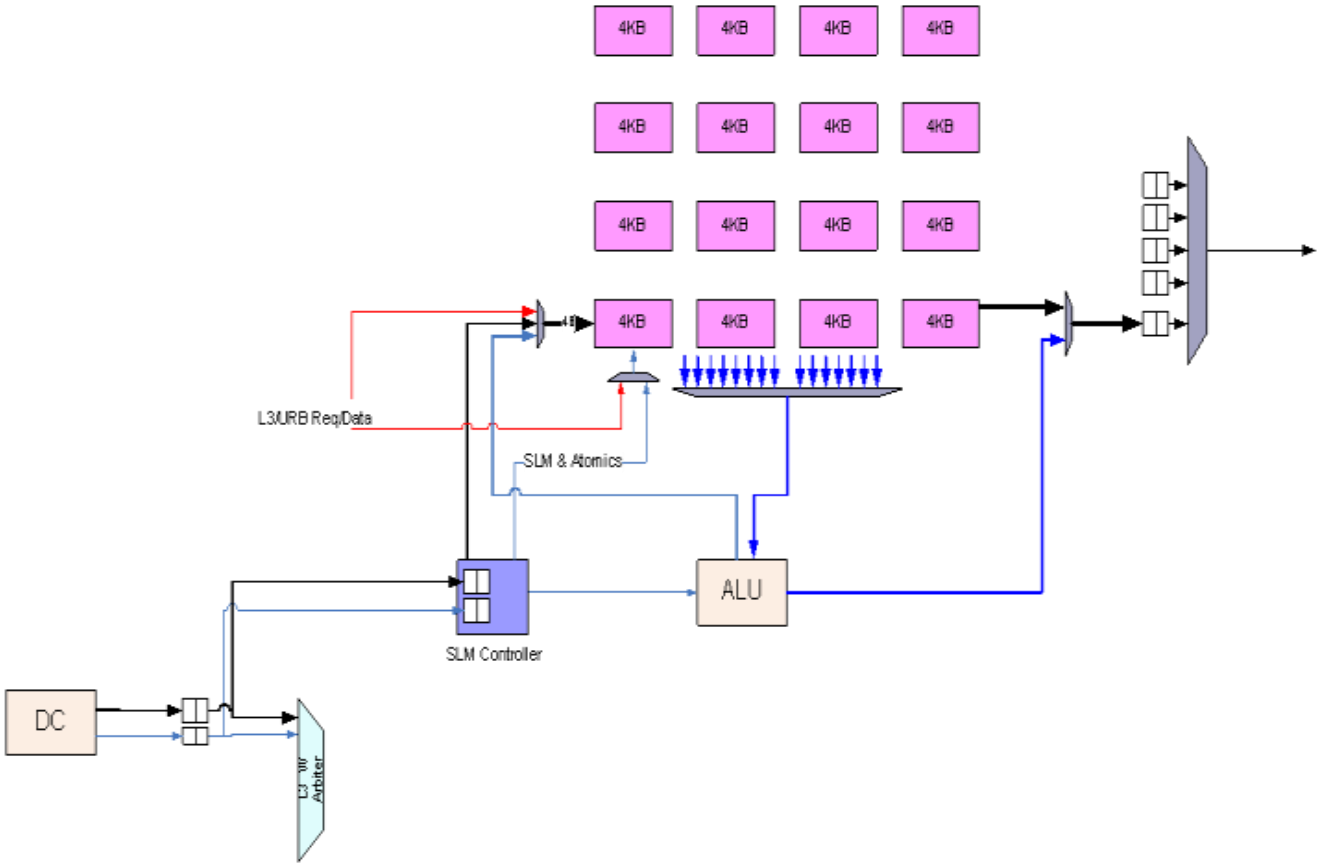


These 16 banks can either be used as URB or used as shared local memory with parallel accesses to all banks. The choice of enabling SLM mode is done through MMIO programming

Bits	Access	Default Value	Description
0	RW/C	0	Enable Shared Local Memory: When set, it enables the use of 2 banks of L3 as shared local memory which allows 64KB of L3 to be banked as 16x4KB and allows independent accesses to all banks within the same clock cycle. **Note:** This mode can only be enabled once L3 content is completely flushed.

SLM is structured as 64KB per L3 bank and each bank allows up to 16 parallel accesses to the 4KB banks. Each 4KB bank is addressed separately where their requests are placed on a 160-bit address bus where each 10-bit correspond to a bank sequentially. Each bank gets addressed with 10 bits and provides 4B access with a total of 4KB per bank. When SLM mode is not enabled, SLM banks are considered a part of

URB.



Save and Restore Requirements

At the pre-emption time for GPGPU the contents of shared local memory will be saved to memory. The flow requires some special handling both in SuperQ and L3 pipeline and will be managed via a small DMA engine which will need to interface to L3 arbiter.

A simple DMA controller within the L3 bank will be designed to generate the accesses that are needed, this block will be used for both save and restore transaction.

Note that GFX virtual buffer given for save and restore has to be 128KB aligned (64KB for GT1).

As part of the process of preemption, the GPGPU controller will ensure there is no more messages from EUs to shared local memory. Once guaranteed, there will be a message to State Arbiter which in turn converted to a message to SLM DMA controller block. From this point on SLM DMA controller will interface with L3 arbiter like any other GFX client with the credit protocol that are defined.

Once all requests are retired to L3 arbiter, the DMA controller will send a completion signal to SARB. State arbiter will wait for both DMA controllers to send their completion signals before sending the response back to GPGPU controller *(note for GT1 – I response is good enough)*

For Save, once the signal is received from State Arbiter to kick-off the save along with the address within the gfx virtual space, DMA controller will start generating following special cycles towards L3 arbiter. Similarly for restore another special request will be sent.

- Save SLM Request
- Restore SLM Request

The request to L3 controller will be 64B size and carry a gfx address similar to any other request that L3 arbiter supports. DMA controller will walk all 64KB address space (1024 entries) via an increment and using the provided address as the base address for request to L3 arbiter.

Once the Save or Restore cycles are all posted to L3 arbiter, DMA controller will signal back to State Arbiter to send the ACK message back to requester.

In L3 arbiter, the save and restore requesters from DMA controller is treated as any other request interface. There is no data involvement so only request queues are needed. A 4 deep structure (ingress FIFO) is provided giving 4 credits to SLM DMA controller to consume.

L3 arbiter will mux in the new ingress FIFO output with HDC read paths and arbitrate Fixed Priority (SLM DMA controller > HDC reads).

The only remaining requirement for L3 arbiter to ensure the flush of this ingress FIFO upon a pipelined Flush received from DC.

In processing save and restore cycles existing flows will be overloaded in SuperQ, both the save and restore request is treated slightly different in SQ FSM.

L3 Cache Error Protection

L3 cache error protection is covered via ECC (secded).

All accesses (including SLM) are subject to ECC protection where single bit errors are fixed silently. Double bit errors are reported via a register structure and communicated by an interrupt to GFX driver. L3 cache HW is additionally capable of stalling execution upon a double bit error; the mechanism is controlled via x7034[9]. If a stall condition is required, the GFX driver is required to initialize the URB prior to execution.

SKL A0 does not have a self-init mechanism for ECC bits between context switches. Lack of self-init shall lead to corruptions due to poisoning of the contents after double bit errors. The w/a that has to be used for A0 stepping is to set: bypass ECC check by programming 0xB120[31]= 1 and 0xB120[0] = 1.

L3 cache and URB have ECC protection on 256b level and SLM has parity based protections. Hardware bases the protection around both URB and SLM to be written before being read, hence there is no automatic initialization. If there is any case where the contents of URB and/or SLM are being read before being written, that could lead to false ECC errors. Hardware does not care about the values read from uninitialized locations, however double-bit ECC errors can trigger an optional "HW lock-up" mode (enabled via MMIO) to protect the contents. Software has to make a choice:

1. For the cases where the workload does not require special double-bit protection, there is no need to enable the "HW lock-up" mode.
2. For the cases where the workload demands an accurate operation and relies on HW detected double-bit errors to lead to HW lock-up, SW will have to initialize the contents of URB or SLM (based on the workload usage model).

L3 Cache and URB

These are the L3 Cache and URB Registers for Gen9. Some of the content is identical to Gen8, with important changes to enhance performance.

LBCF Registers

L3SQCREG1 - L3 SQCRegisters1

L3SQCregisters2

L3SQCregisters3

L3ControlRegister1

L3SLMRegister

L3SQCregister4

SCRATCH1

LTCDconfigregister

LBSconfigbits

L3BankStatus

LBCFconfigsavemsg

LNCF Registers

ArbiterControlRegister

SCRATCHforLNCFunit

Eventselectionandbasecounters

EventSelectionandBaseCounters1

LPFCcontrolregister

LNCFconfigsavemsg

LPFC Registers

PMflushandslmsaverestore

GlobalInvalidationRegister

PowercontextSaveRegisterforLPFC

FirstBufferSizeandStart

SecondBufferSize

ErrorReportingRegister

FramecountandDrawcallnumber

SaveTimer

Masterstarttimer

EU Overview

The GEN 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 GEN 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 GEN native instructions may be required. In addition, there are many specific capabilities that accelerate media applications. The following feature list summarizes the GEN 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 executed in multiple cycles over a SIMD execution pipeline.
- Most GEN instructions have three operands. Some instructions have additional implied source or destination operands. Some instructions have explicit dual destinations.
- Region-based register addressing.
- Direct or indirect (indexed) register addressing.
- Scalar or vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

CoIssue/Dual Issue:

The EM Pipe has been expanded to handle general integer and all the float operations that FPU pipe can execute as well.

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).

Introduced an additional thread scheduling policy. 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.

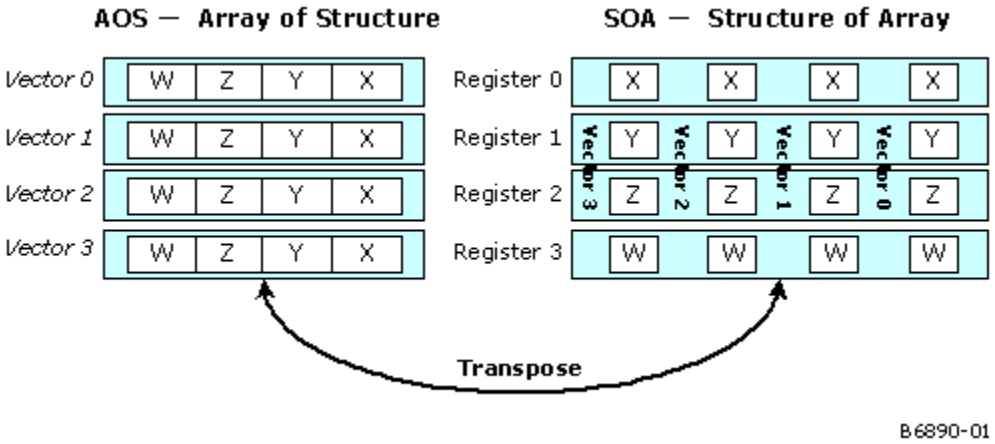
Primary Usage Models

In describing the usage models of the GEN 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 GEN 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



GEN 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*mxn*, where *m* is the size of the vector and *n* is the number of concurrent program flows executed in SIMD.

Execution with AOS data structures:

- **SIMD4** (short for SIMD4x1) indicates that a SIMD instruction operates on 4-element vectors storedin registers. There is one program flow.
- **SIMD4x2** indicates that a SIMD instruction operates on a pair of 4-element vectors in registers. There are effectively two programs running side by side with one vector per program.

Execution with SOA data structures, also referred to as "channel serial" execution, mostly uses:

- **SIMD8** (short for SIMD1x8) indicates a SIMD instruction based on the SOA data structure where one register contains one data element (the same one) for each of 8 vectors. Effectively, there are 8 concurrent program flows.
- **SIMD16** (short for SIMD1x16) indicates that a SIMD instruction operates on a pair of registers that contain one data element (the same one) for each of 16 vectors. SIMD16 has 16 concurrent program flows.

SIMD4 Mode of Operation

With a register mapping of src0 to doublewords 0-3 of *r2*, src1 to doublewords 4-7 of *r2* and dst to doublewords 0-3 of *r3*, the example 3D graphics API Shader instruction can be translated into the following GEN instruction:

add (4) r3<4>.xyz:f r2<4>.yzwx:f r2.4<4>.zwxy:f {NoMask}

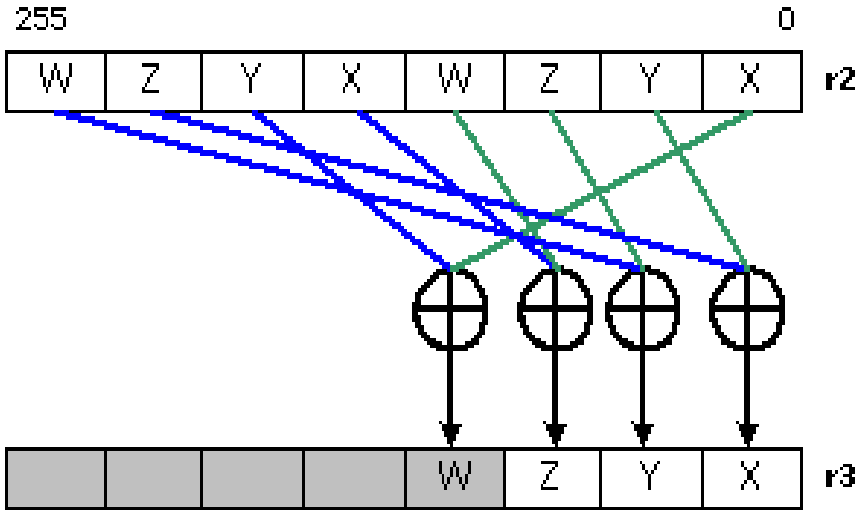
Without diving too much into the syntax definition of a GEN instruction, it is clear that a GEN instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar to the term SIMD Width used in the literature. Each operand is described by the register region parameters such as '<4>' and data type (e.g. "*:f*"). These will be detailed in the SIMD8 Mode of Operation section. The instruction option field, {NoMask}, ensure that the execution occurs for the execution channels shown in the instruction, instead of, possibly, being masked out by the conditional masks of the thread (See Instruction Summary chapter for definition of *MaskCtrl* instruction field).

The operation of this GEN instruction is illustrated in the following figure. In this example, both source operands share the same physical GRF register r2. The two are distinguished by the subregister number. The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 are not modified. In particular, doublewords 4-7 are unchanged as the execution size is 4; doubleword 3 is unchanged due to the destination mask setting.

In this mode of operation, there is only one program flow – any branch decision will be based on a scalar condition and apply to the whole vector of four elements. Option {*NoMask*} ensures that the instruction is not subject to the masks. In fact, most of the instructions in a thread should have {*NoMask*} set.

Even though the execution only performs four parallel add operations, the GEN instruction still executes in 2 cycles (with no useful computation in the second cycle).

A SIMD4 Example



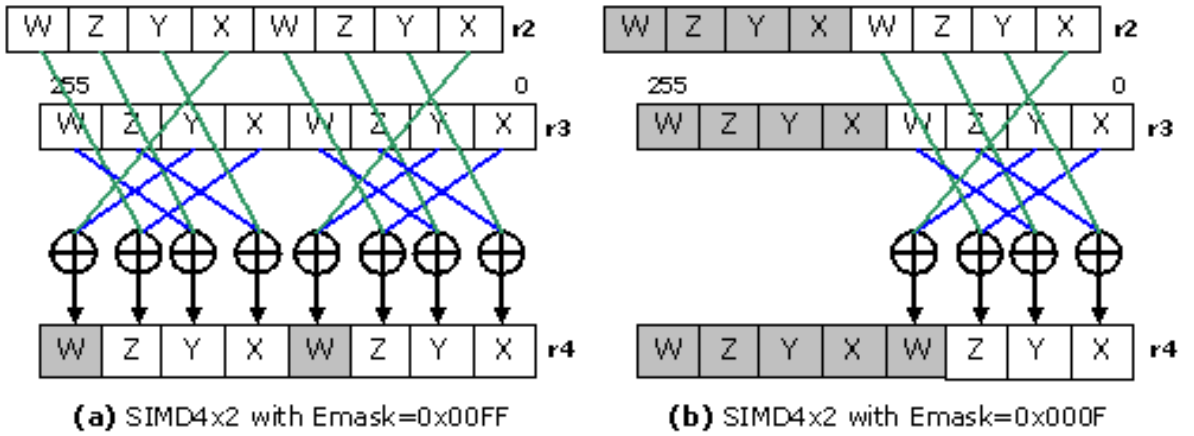
SIMD4x2 Mode of Operation

In this mode, two corresponding vectors from the two program flows fill a GEN register. With a register mapping of src0 to r2, src1 to r3 and dst to r4, the example 3D graphics API Shader instruction can be translated into the following GEN instruction:

add (8) r4<4>.xyz:f r2<4>.yxzw:f r3<4>.zwxy:f

This instruction is subject to the execution mask, which initiated from the dispatch mask. If both program flows are available (e.g. Vertex Shader executed with two active vertices), the dispatch mask is set to 0x00FF. The operation of this GEN instruction is illustrated in *SIMD4x2 Mode of Operation* (a). The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 (doublewords 3 and 7) are unchanged due to the destination mask setting. If only one program flow is available (e.g. the same SIMD4x2 Vertex Shader with only one active vertex), the dispatch mask is set to 0x000F. The operation of the same instruction is shown in *SIMD4x2 Mode of Operation* (b).

SIMD4x2 Examples with Different Emasks

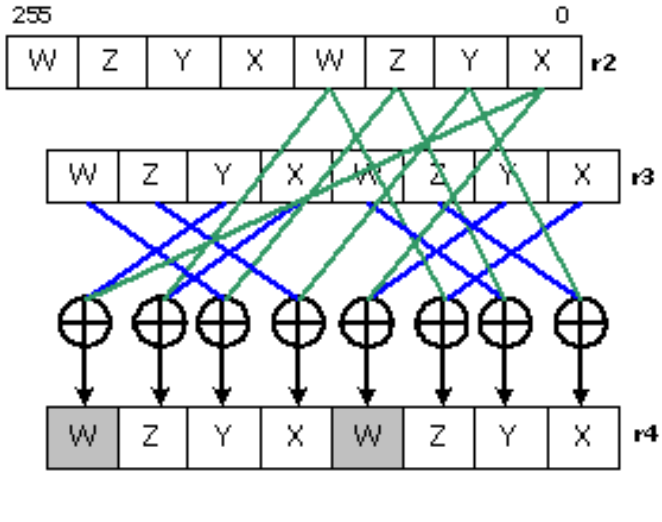


The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2, which is shared by the two program flows. The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

*add (8) r4<4>.xyz:f r2<**0**>.yzwx:f r3<4>.zwxy:f*

The only difference here is that the vertical stride of the first source is 0. The operation of this GEN instruction is illustrated in *SIMD4x2 Mode of Operationicl*.

A SIMD4x2 Example with a Constant Vector Shared by Two 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 GEN 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 GEN instructions:

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

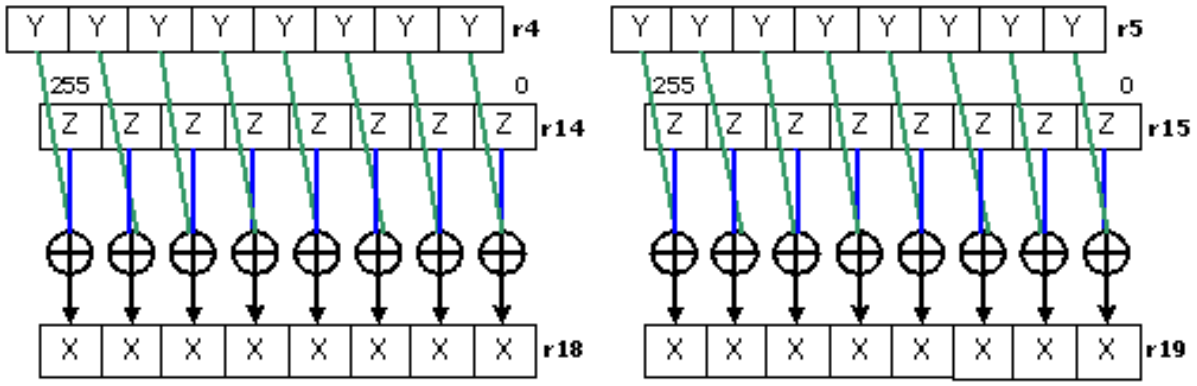
add (16) r20<1>:f r6<8;8,1>:f r16<8;8,1>:f// dst.y = src0.z + src1.w

add (16) r22<1>:f r8<8;8,1>:f r10<8;8,1>:f // dst.z = src0.w + src1.x

The three GEN instructions correspond to the three enabled destination masks As there is no output for the W elements of dst, no instruction is needed for that element. The first instruction inputs the Y elements of src0 and the Z elements of src1 and outputs the X elements of dst. The operation of this instruction is shown in *SIMD16 Mode of Operation*.

With more than one program flow, the above instructions are also subject to the execution mask. The 16-bit dispatch mask is partitioned into four groups with four bits each. For Pixel Shader generated by the Windower, each 4-bit group corresponds to a 2x2 pixel subspan. If a subspan is not valid for a Pixel Shader instance, the corresponding 4-bit group in the dispatch mask is not set. Therefore, the same instructions can be used independent of the number of available subspans without creating bogus data in the subspans that are not valid.

A SIMD16 Example



Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS format). The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

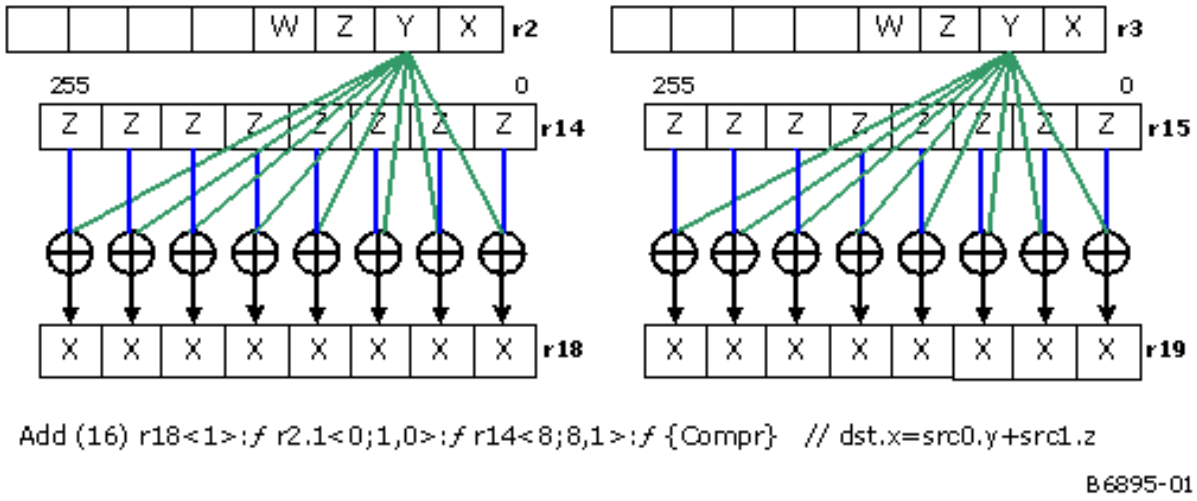
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr}// dst.x = src0.y + src1.z

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 GEN instruction is illustrated in *SIMD16 Mode of Operation*.

Another SIMD16 Example with an AOS Shared Constant



SIMD8 Mode of Operation

Each compressed instruction has two corresponding native instructions. Taking the example instruction shown in *SIMD16 Mode of Operation*, 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, some time 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.

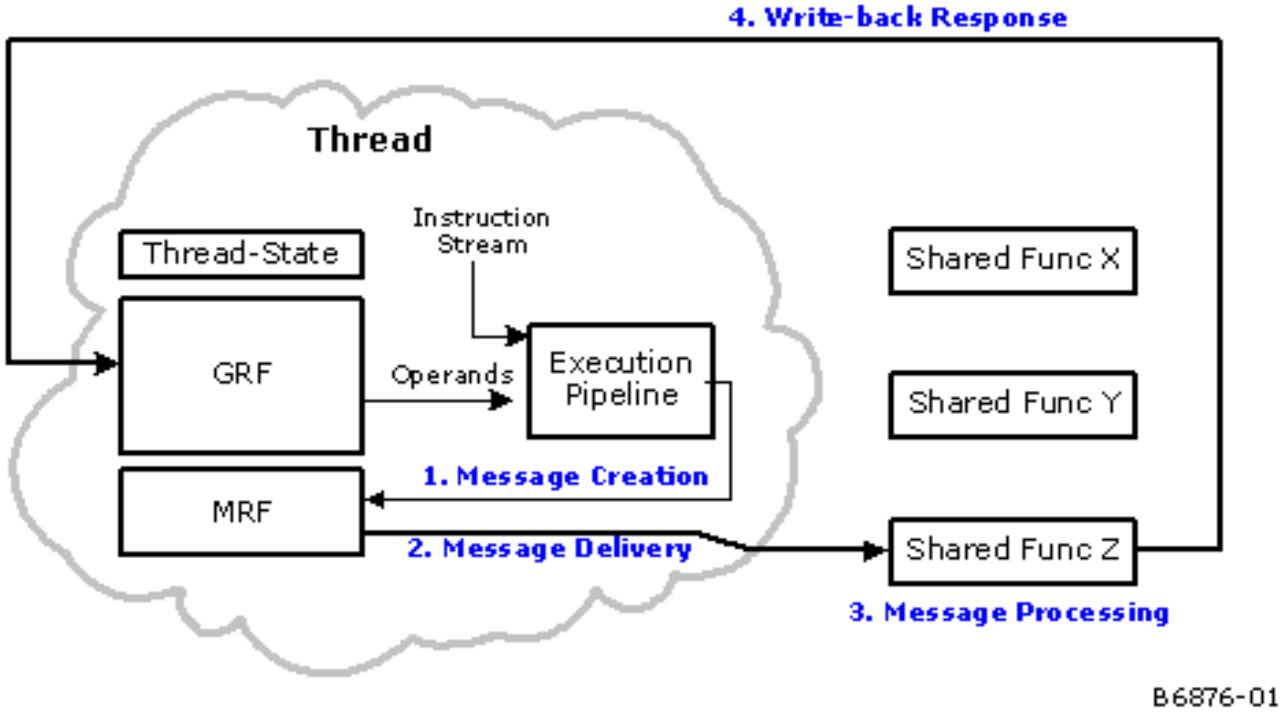
The information transmitted in a message falls into two categories:

- **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 *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 Gen4 execution unit and message passing infrastructure do not interpret this field in any way to determine if writeback data is to be expected. Instead explicit fields in the 'send' instruction to the execution unit the starting GRF register and count of returning data. The execution unit uses this information to set in-flight bits on those registers to prevent execution of any instruction which uses them as an operand until the register(s) is(are) eventually written in response to the message. If a message is not expected to return data, the 'send' instruction's writeback destination specifier (<post_dest>) must be set to 'null' and the response length field of <desc> must be 0 (see 'send' instruction for more details).

The writeback data, if called for, arrives as a series of register writes to the GRF at the location specified by the starting GRF register and length as specified in the 'send' instruction. As each register is written

back to the GRF, its in-flight flag is cleared and it becomes available for use as an instruction operand. If a thread was suspended pending return of that register, the dependency is lifted and the thread is allowed to continue execution (assuming no other dependency for that thread remains outstanding).

Message Delivery Ordering Rules

All messages between a thread and an individual shared function are delivered in the ordered they were sent. Messages to different shared functions originating from a single thread may arrive at their respective shared functions out of order.

The writebacks of various messages from the shared functions may return in any order. Further individual destination registers resulting from a single message may return out of order, potentially allowing execution to continue before the entire response has returned (depending on the dependency chain inherent in the thread).

Execution Mask and Messages

The Gen4 Architecture defines an Execution Mask (EMask) for each instruction issued. This 16b 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 16b 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 Extended Math shared function observes this field and performs the specified operation only on the operands with enabled channels, while the DataPort writes to the render cache ignore this field completely, instead using the pixel mask included in-band in the message payload to indicate which channels carry valid data.

End-Of-Thread (EOT) Message

The final instruction of all threads must be a *send* instruction that signals 'End-Of-Thread' (EOT). An EOT message is one in which the EOT bit is set in the *send* instruction's 32b <desc> field. When issuing instructions, the EU looks for an EOT message, and when issued, shuts down the thread from further execution and considers the thread completed.

Only a subset of the shared functions can be specified as the target function of an EOT message, as shown in the table below.

Target Shared Functions supporting EOT messages	Target Shared Functions not supporting EOT messages
PixelPort, URB, ThreadSpawner	DataPort, 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.

All threads generated by Thread Spawner must send a message to Thread Spawner to explicity signal end-of-thread.

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 Gen4 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 a Gen4 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:

Description
Software prefetching techniques may be beneficial for long latency data fetches (i.e. issue a load early in the thread for data that is required late in the thread).

Message Description Syntax

All message formats are defined in terms of DWords (32 bits). The message registers in all cases are 256 bits wide, or 8 DWords. The registers and DWords within the registers are named as follows, where n is the register number, and d is the DWord number from 0 to 7, from the least significant DWord at bits [31:0] within the 256-bit register to the most significant DWord at bits [255:224], respectively. 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 *Message Errors* 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	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

Error	Outcome
Incorrect message length	long or too short, and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through and interrupt mechanism then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.
Incorrect response length	Case: too few registers specified – the thread may proceed with execution prior to all the data returning from the shared function, resulting in the thread operating on bad data in the GRF. Case: too many registers specified – the message response does not clear all the registers of the destination. In this case, if the thread references any of the residual registers, it may hand and result in an eventual time-out.
Improper use of End-Of-Thread (EOT)	Any 'send' instruction which specifies EOT must have a 'null' destination register. The EU enforces this and, if detected, will not issue the 'send' instruction, resulting in a hung thread and an eventual time-out. The 'send' instruction specifies that EOT is only recognized if the <desc> field of the instruction is 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.
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

GEN 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, GEN 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 register, r0 through r127.

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

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.

Description
GRF registers are directly indicated by a unique register file encoding.

Summary of Architecture Registers

Register Type (RegNum [7:4])	Register Name	Register Count	Description
0000b	*null*	1	Null register
0001b	*a0.#*	1	Address register
0010b	*acc#*	10	Accumulator register
0011b	*f#.#*	2	Flag register
0100b	*ce#*	1	Channel Enable register
0101b	*msg#*	32	Message Control Register
0110b	*sp*	1	Stack Pointer Register
0111b	*sr0.#*	2	State register

Register Type (RegNum [7:4])	Register Name	Register Count	Description
1000b	*cr0.#*	1	Control register
1001b	*n#*	2	Notification Count register
1010b	*ip*	1	Instruction Pointer register
1011b	*tdr*	1	Thread Dependency register
1100b	*tm0*	2	TimeStamp register
1101b	*fc#.#*	39	Flow Control register
1110b	*Reserved*		Reserved

Programming Note	
Context:	ARF Registers Overview
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.	

Programming Note	
Context:	ARF Registers Overview
In general an ARF register can be dst (destination) or src0 (source 0, first source operand) for an instruction. Depending on the register and the instruction, other restrictions may apply.	

Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF.

The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to generate the destination write mask at byte level.

Subregister fields of an ARF register may not all be populated (indicated by the subregister being indicated as reserved). Writes to unpopulated subregisters are dropped; there are no side effect. Reads from unpopulated subregisters, if not specified, return unpredictable data.

Some ARF registers are read-only. Writes to read-only ARF registers are dropped and there are no side effects.

Null Register

Null Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control?	N/A
Indexable?	No

The null register is a special encoding for an operand that does not have a physical mapping. It is primarily used in instructions to indicate non-existent operands. Writing to the null register has no side effect. Reading from the null register returns an undefined result.

The null register can be used where a source operand is absent. For example, for a single source operand instruction such as MOV or NOT, the second source operand src1 must be a null register.

When the null register is used as the destination operand of an instruction, it indicates the computed results are not stored in any registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, updates to the selected flag register also occur. In this case, the register region fields of the 'null' operand are valid.

Another example use is to use the null register as the posted destination of a *send* instruction for data write to indicate that no write completion acknowledgement is required. In this case, however, the register region fields are still valid. The null register can also be the first source operand for a send instruction indicating the absent of the implied move. See the *send* instruction for details.

Address Register

Address Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0001b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	16

Attribute	Value
Element Size:	16 bits
Element Type:	UW or UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	N/A
Indexable?	No

Description
There are sixteen address subregisters forming a 16-element vector. Each address subregister contains 16 bits. Address subregisters can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the *send* instruction.

Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000b = a0 All other encodings are reserved.	When register a0 or subregisters in a0 are used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned. 00000b = **a0.0:uw** 00010b = **a0.1:uw** 00100b = **a0.2:uw** 00110b = **a0.3:uw** 01000b = **a0.4:uw** 01010b = **a0.5:uw** 01100b = **a0.6:uw** 01110b = **a0.7:uw** 10000b = **a0.8:uw** 10010b = **a0.9:uw** 10100b = **a0.10:uw** 10110b = **a0.11:uw** 11000b = **a0.12:uw** 11010b = **a0.13:uw** 11100b = **a0.14:uw**

RegNum[3:0]	SubRegNum[4:0]
	11110b = **a0.15:uw** All other encodings are reserved. However, when register a0 or subregisters in a0 are explicit source and/or destination registers, other data types are allowed as long as the register region falls in the 128-bit range.

Address Register Fields

DWord	Bits	Description
7	31:16	**Address subregister a0.15:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.14:uw**. Follows the same format as **a0.2**.
6	31:16	**Address subregister a0.13:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.12:uw**. Follows the same format as **a0.2**.
5	31:16	**Address subregister a0.11:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.10:uw**. Follows the same format as **a0.2**.
4	31:16	**Address subregister a0.9:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.8:uw**. Follows the same format as **a0.2**.
3	31:16	**Address subregister a0.7:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.6:uw**. Follows the same format as **a0.2**.
2	31:16	**Address subregister a0.5:uw**. Follows the same format as a0.3.
	15:0	**Address subregister a0.4:uw**. Follows the same format as a0.2.
1	31:16	**Address subregister a0.3:uw**. This field, with only the lower 12 bits populated can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.2:uw**. This field, with only the lower 12 bits populated can be used as an unsigned integer for register-indirect register addressing. Format: U12
0	31:16	**Address subregister a0.1:uw**. This field can be used for register-indirect register addressing or serve as message descriptor for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the higher 16 bits of <desc>.

DWord	Bits	Description
		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 for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the lower 16 bits of <desc>. Format: U12 or U16.

When used as a source or destination operand, the address subregisters can be accessed individually or as a group. In the following example, the first instruction moves 8 address subregisters to the first half of GRF register r1, the second instruction replicates a0.4:uw as an unsigned word to the second half of r1, the third instruction moves the first 4 words in r1 into the first 4 address subregisters, and the fourth instruction replicates r1.4 as a unsigned word to the next 4 address subregisters.

```
mov (8) r1.0<1>:uw a0.0<8;8,1>:uw // r1.n = a0.n for n = 0 to 7 in words
mov (8) r1.8<1>:uw a0.4<0;1,0>:uw // r1.m = a0.4 for m = 8 to 15 in words
mov (4) a0.0<1>:uw r1.0<4;4,1>:uw // a0.n = r1.n for n = 0 to 3 in words
mov (4) a0.4<1>:uw r1.4<0;1,0>:uw // a0.m = r1.4 for m = 4 to 7 in words
```

When used as the register-indirect addressing for GRF registers, the address subregisters can be accessed individually or as a group. When accessed as a group, the address subregisters must be group-aligned. For example, when two address subregisters are used for register indirect addressing, they must be aligned to even address subregisters. In the following example, the first instruction is legal. However, the second one is not. As ExecSize = 8 and the width of src0 is 4, two address subregisters are used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address subregisters are a0.2:uw and a0.3:uw. The two align to a DWord group in the address register. However, the two address subregisters for the second instruction are a0.3:uw and a0.4:uw. They are not DWord-aligned in the address register and therefore violate the above mentioned alignment rule.

```
mov (8) r1.0<1>:d r[a0.2]<4,1>:d // a0.2 and a0.3 are used for src1
mov (8) r1.0<1>:d r[a0.3]<4,1>:d // ILLEGAL use of register indirect
```

Programming Note	
Context:	ARF Registers
Implementation restriction: When used as the source operand <desc> for the send instruction, only the first dword subregister of a0 register is allowed (i.e. a0.0:ud, which can be viewed as the combination of a0.0:uw and a0.1:uw). In addition, it must be of UD type and in the following form <desc> = a0.0<0;1,0>:ud.	

Programming Note	
Context:	ARF Registers
Restriction: Elements a0.0 and a0.1 have 16 bits each, but the rest of the elements (a0.2:uw through a0.7:uw) only have 12 bits populated each. 12-bit precision supports full indirect-addressing capability for the largest GRF register range. Software must observe the asymmetry of the implementation. When a0.0:uw and a0.1:uw are the source or	

Programming Note	
Context:	ARF Registers
destination, full 16-bit precision is preserved. However, when a0.2:uw to a0.7:uw are the destination, the high 4 bits for each element are dropped; when they are the source, hardware inserts zero to the high 4 bits for each element.	

Programming Note	
Context:	ARF Registers
Performance Note: There is only one scoreboard for the whole address register. When a write to some subregisters is in flight, hardware stalls any instruction writing to other subregisters. Software may use the destination dependency control {NoDDChk, NoDDClr} to improve performance in this case. Similarly, when a write to some subregisters is in flight, hardware stalls any instruction sourcing other subregisters until the write retires.	

Accumulator Registers

Accumulator Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	10
Default Value:	None
Normal Access:	RW

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.

Accumulator Registers
There are 10 accumulator registers. The accumulator registers are of two types.

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}
```

- It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). The result of such an instruction is unpredictable.
-
- 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.

Exceptions

When OWords of accumulators are accessed, the source and destination OWords may be different. For example, the following instructions are allowed:
`mov (4) r10.4<1>:f acc0.0<1>:f`
`add (4) r10.0<1>:f acc0.4<1>:f r11.0<1>:f`
`mov (8) r10.8<1>:uw acc0.0<1>:uw`
`add (8) r10.0<1>:uw acc0.8<1>:uw r11.0<1>:uw`

Exceptions
The source and destination datatypes MUST be the same for such access of accumulator. If destination is contained within one register, source must also be contained within one accumulator register.

Implementation Precision Restriction: As there are only 64 bits per channel in DWord mode (D and UD), it is sufficient to store the multiplication result of two DWord operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The DWord multiplication result is then 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.

Implementation Precision Restriction: As there are only 33 bits per channel in Word mode (W and UW), it is sufficient to store the multiplication result of two Word operands with and without source modifier as the result is up to 33 bits. Integers are stored in accumulator in 2's complement form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into a risk of overflowing. When overflow occurs, only modular addition can generate a correct result. But in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UD (FFFFFFFFh) with D (FFFFFFFFh) results in 1FFFFFFFEh. The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.

Accumulator Channel Precision

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
DF	acc0	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 *mul*, *mach*, and *shr* (or *mov*).
W (UW)	acc0	16	33	When the internal execution data type is word integer, each accumulator register contains 16 channels of (extended) word integer

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
				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.
B (UB)	N/A	N/A	N/A	Not supported data type.

Accumulators
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. Each of these accumulator registers are defined as 256-bit registers having 8 DWords. These may be accessed explicitly or implicitly.

- These registers may be accessed explicitly only by a *mov* 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.

  Example:

```
// Move 256 bits from acc2 to r10. Just low two bits of each DWord are valid:

mov (8) r10:ud acc2:ud

// Move 256 bits from r10 to acc2. Just low two bits of each DWord are updated:
mov (8) acc2:ud r10:ud
```

- These registers are accessed implicitly by three opcodes defined for the macro operations. **Note:** These macro operations are defined under the *math* opcode section. The macro descriptions also define the restrictive implicit uses of these registers.

Description

Implicit access across accumulator registers is required for each source operand for these macro instructions. These opcodes are accessed in Align16 mode only. The Channel Select bits in the instruction are used to implicitly address the different accumulators for each source. Similarly the Channel Enable bits are used to implicitly address the accumulators for destination. The noacc value is specified when no write to accumulator is required; think of it as a null.

Encoding	Accumulator Register
00000001b	acc3
00000010b	acc4
00000011b	acc5

Description	
00000100b	acc6
00000101b	acc7
00000110b	acc8
00000111b	acc9
00001000b	noacc

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:	32 bits
Element Type:	UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	Yes
Indexable?	No

Description
There are two flag registers, f0 and f1.

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.

Description
Flags can be split to halfs, quarters, or eighths using the QtrCtrl and NibCtrl instruction fields. Those fields affect the selection of flags for predication and conditional modifiers, but do not affect reading or writing flags as explicit instruction operands.

The values held in the individual bits of a flag register are the result of the most recent instruction with a conditional modifier and specifying that flag register. For example:

```
add.nz.f0.0 ...
```

Updates flag subregister f0.0 with the per-channel results of the not zero condition.

The flag register has per-bit write enables. When being updated as the secondary destination associated with a conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. Other bits in the flag subregister are unchanged.

Flag registers and subregisters can also be explicit source or destination operands.

The *sel* instruction does not update flags.

Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000b = f0:ud 0001b = f1:ud Other encodings are reserved.	00000b = f*n*.0:uw 00010b = f*n*.1:uw Other encodings are reserved.

Description
Reference an entire flag register as f0:ud or f1:ud. Reference the flag subregisters as f0.0:uw, f0.1:uw, f1.0:uw, and f1.1:uw.

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**.

RegNum[3:0]	SubRegNum[4:0]
All other encodings are reserved.	All other encodings are reserved.

Channel Enable Register Fields

DWord	Bits	Description
0	31:0	**Channel Enable Register ce:ud** 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:	8
Default Value:	None
Normal Access:	RW
Elements:	2
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Word
Write Mask Granularity:	Word

Register and SubRegister Numbers for Message Control Registers

RegNum[3:0]	SubRegNum[4:0]
0000b - 0111 = msg0 - msg7 All other encodings are reserved.	MBZ

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 behaviour.

Each thread has 8 registers. The granularity of access is always one full register, i.e., 256b. 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. Access mode is ignored when reading/writing these registers.

These registers must be accessed in order, i.e., reads/writes are in order from msg0 to msg7.

Programming Note

Context:	Message Control Registers
Message Control Registers must never be saved or restored	

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 integer (or as a QWord on BDW+). 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	63:48	**Reserved**. MBZ.
	47:0	**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. *sp = [scratch space pointer] + [scratch space] - 1* 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 exclusive for the thread.
2..3	63:48	**Reserved**. MBZ.
	47:0	**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. *sp_limit = [scratch space pointer] + [stack space limit]* 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.

State Register

State Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0111b
Number of Registers:	2
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 – 01100b All other encodings are reserved.
0001b = sr1 All other encodings are reserved.	Valid encoding range: 00000b 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.
	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.
	22:19	**Reserved**. MBZ.
	18:16	**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

DWord	Bits	Description
		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.
	7:3	**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.
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
3 (sr0.3:ud)	31:0	**Vector Mask** (VMask). This 32-bit field contains, for each 4-bit group, the OR of the corresponding 4-bit group in the dispatch mask. This field is used by hardware to initialize the mask register. Format: U32
0 (sr1.0:ud)	31:0	**Hardware Defined State Register.** The contents of these register are hardware defined and are required only for handling page-fault. These bits are saved and restored by SIP when threads are pre-empted. Writes to these registers must follow the sequence described in 'send' instruction for the correct behavior of hardware.
1 (sr1.1:ud)	31:0	**Hardware Defined State Register.** Same as sr1.0
2 (sr1.2:ud)	31:0	**Hardware Defined State Register.** Same as sr1.0
3 (sr1.3:ud)	31:0	**Hardware Defined State Register.** Same as sr1.0

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 set the thread control field to *'switch'* for 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 set *'switch'*, 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
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 a GEN 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 = cr0 All other encodings are reserved.	00000b = **cr0.0:ud**. It contains general thread control fields. 00100b = **cr0.1:ud**. It contains exception status and control. 01000b = **cr0.2:ud**. It contains AIP. All other encodings are reserved.

Control Register Fields

DWord	Bits	Description
0	31	**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. This bit is initialized to 0. 0 = The thread is in normal state. 1 = The thread is in exception state.
	30:16	**Reserved.** MBZ.
	14:13	**Reserved.** MBZ.
	12	**IEEE Float to Integer Rounding Mode.** This bit determines how rounding modes are handled in float to integer conversion operation. This bit is initialized to 0 during Thread Dispatch. This bit must be set for IEEE compliant float to integer conversion operation. 0= The result of float to integer conversion operation is with RTZ rounding mode. 1= The result of float to integer conversion operation is with rounding mode programmed in cr0.0[5:4].

DWord	Bits	Description
	11	**IEEE MinMax.** This bit determines how SNAN is handled in min/max operations. This bit is initialized to 0 during Thread Dispatch. This bit must be set for IEEE compliant min/max operation. 0 = The result of min/max is a non-SNAN source. 1 = The result of min/max is the SNAN source.
	10	**Half Precision Denorm Mode.** This bit determines how denormal numbers are handled for the HF (Half Float) type. This bit is initialized to 0 during Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results.
	9	**IEEE Exception Trap Enable.** This bit enables trapping IEEE exception flags. This control bit may updated by software. It is initially zero on thread load. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exceptions field of sr0.1. Note that IEEE floating-point exceptions do *not* transfer control to any handler. 0 = IEEE Exception flags are NOT trapped. 1 = IEEE Exception flags are trapped.
	7	**Single Precision Denorm Mode.** This bit determines how denormal numbers are handled for the F (Float) type when using the IEEE floating-point mode. It is ignored in the ALT floating-point mode, which always flushs denorms. This bit is initialized by Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results.
	6	**Double Precision Denorm Mode.** This bit determines how denormal numbers are handled for the DF (Double Float) type. It is initialized by Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results.
	5:4	**Rounding Mode.** This field specifies the FPU rounding mode. It is initialized by Thread Dispatch. 00b = Round to Nearest or Even (RTNE) 01b = Round Up, toward +inf (RU) 10b = Round Down, toward -inf (RD) 11b = Round Toward Zero (RTZ)

DWord	Bits	Description
	3	**Vector Mask Enable (VME).** This bit indicates DMask or Vmask should be used by EU for execution. This bit is set by the Thread Dispatch. 0: Use Dispatch Mask (DMASK) 1: Use Vector Mask (VMASK)
	2	**Single Program Flow (SPF).** Specifies whether the thread has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). This bit affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This bit is initialized by the Thread Dispatch. 0: Multiple Program Flows 1: Single Program Flow **Programming Restrictions:** Only H1/Q1/N1 are allowed in SPF mode. Power Optimization: If an entire shader does not do SIMD branching, the driver can set the SPF bit to 1 to save power in HW.
	1	**Accumulator Disable.** This bit controls the update of the accumulator by the instruction field AccWrCtrl. If this bit is cleared, the accumulator is updated for all instructions with AccWrCtrl enabled. If set, the accumulator is disabled for all update operations, maintaining its value prior to being disabled. Setting this bit has no effect if the accumulator is the explicit destination operand for an instruction. This bit is initialized to 0. 0: Enable accumulator update. 1: Disable accumulator update. **Usage Notes:** This control bit is primarily designed for the System Routine. That routine is not expected to use the accumulator, though it may need to use instructions that implicitly update the accumulator. To use such instructions in the System Routine, but still preserve the accumulator contents on returning to the application kernel, the System Routine would either (a) save and restore the accumulator, or (b) prevent the accumulator from being unintentionally modified. This control bit has been added for the latter method. Software has the option to limit the setting of this control bit to strictly within the System Routine. If, by convention, this bit is clear within application kernels, the System Routine can simply set the bit upon entry and clear it before returning control to the application kernel. This usage model would not necessarily require cr0.0 to be saved/restored in the System Routine. However, if by convention application kernels are permitted to set this bit, then the System Routine is required to preserve the content of this bit.

DWord	Bits	Description
	0	**Single Precision Floating Point Mode (FP Mode).** This bit specifies whether the current single-precision floating-point operation mode is IEEE mode (IEEE Standard 754) or the ALT (alternative mode). This bit does not affect the floating-point mode used for other floating-point data types. This bit is also forwarded on the message sideband for all out-going messages, for example, to control the floating-point mode of the Sampler. Software may modify this bit to dynamically switch between the two floating-point modes. This bit is initialized by Thread Dispatch. 0 = IEEE floating-point mode for the F (Float) type. 1 = ALT (alternative) floating-point mode for the F (Float) type.
1	30	**External Halt Exception Status and Control**. This bit indicates the External Halt exception. It is set by EU hardware on receiving the broadcast External Halt signal. The System Routine should reset this bit before returning to an application routine to avoid infinite loops. This bit may be set or cleared by software. This bit is initialized to 0.
	29	**Software Exception Control**. This bit is the control bit for software exceptions. Setting this bit to 1 in an application routine causes an exception. Clearing this bit in an application routine has no effect. Upon entering the system routine, the hardware maintains this bit as 1 to signify a software exception. The System Routine should reset this bit before returning to an application routine. This bit may be set or cleared by software. This bit is initialized to 0.
	28	**Illegal Opcode Exception Status**. This bit, when set, indicates an illegal opcode exception. The exception handler routine normally does not return back to the application thread upon an illegal opcode exception. Leaving this bit set has no effect on hardware; if system software adversely returns to an application routine leaving this bit set, it doesn't cause any exception. This bit should not be set by software or left set by the system routine to avoid confusion. This bit is initialized to 0.
	27	**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. This bit is initialized to 0.
	25	**Context Save Status.** This bit when set, indicates a Context Save process has been initiated. The system routine must reset this bit after saving the context to terminate the thread.
	24	**Context Restore Status.** This bit when set, indicates a Context Restore process has been initiated. The system routine must reset this bit after restoring the context. The reset of this bit is required before invoking application routine.
	23:16	**Reserved**. MBZ.

DWord	Bits	Description
	13	**Software Exception Enable**. This bit enables or disables the software exception. Enabling or disabling this bit may allow host software to turn on/off certain features (such as profiling) without changing the kernel program. This bit is initialized by the Thread Dispatcher. Format = ENABLED: 0: Disabled 1: Enabled
	12	**Illegal Opcode Exception Enable**. This bit specifies whether the illegal opcode exception is enabled or not. The Illegal opcode exception includes illegal opcodes and undefined opcodes, caused by bad programs or run-time data corruption. This bit is initialized by the Thread Dispatcher. Software should normally assign this bit in the interface descriptor. Even though this mechanism is provided to disable the illegal opcode exception, it should be used with extreme caution. Format = ENABLED: 0: Disabled 1: Enabled
	11	**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. This bit is initialized by the Thread Dispatcher. Software should normally assign this bit in the interface descriptor. Format = ENABLED: 0: Disabled 1: Enabled
	10:0	**Reserved.** MBZ.
2 (cr0.2:ud)	31:3	**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.

Exception Type	AIP Value
External Halt	N/A (1)

DWord	Bits	Description
		<table><tr><td>Software Exception</td><td>IP + 1</td></tr><tr><td>Illegal Opcode</td><td>IP</td></tr></table> (1) External Halt exception is asynchronous and not associated with an instruction. When the System Routine changes the 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.
	2:0	**Reserved.** MBZ.

Programming Note	
Context:	
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 set the thread control field to *'switch'* for 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 set *'switch'*, subsequent instructions may have undefined results.	

Notification Registers

Notification Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1001b
Number of Registers:	3
Default Value:	No
Normal Access:	RO (RW – Context save/restore only)
Elements:	3
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

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.

Workaround	
Context:	Notification Registers
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 non zero, 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.

Register and Subregister Numbers for Notification Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = n0 All other encodings are reserved.	00000b = n0.0:ud 00100b = n0.1:ud 01000b = n0.2:ud All other encodings are reserved.

Notification Register 0 Fields

DWord	Bits	Description
0	31:16	**Reserved**. MBZ.
	15:0	**Thread to Thread Notification Count**. This register is used by the WAIT 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 WAIT instruction. See the WAIT instruction for details. Format: U16

Notification Register 1 Fields

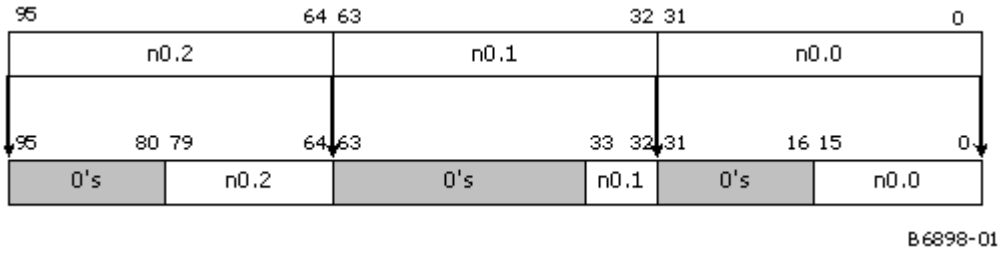
DWord	Bits	Description
0	31:1	**Reserved**. MBZ.
	0	**Host to Thread Notification**. This register is used by the WAIT instruction for host-to-thread

DWord	Bits	Description
		synchronization via MMIO registers. Format: U1

Notification Register 2 Fields

DWord	Bits	Description
0	31:16	**Reserved**. MBZ.
	15:0	**Thread to Thread Notification Count**. This register is used by the WAIT 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 WAIT instruction. See the WAIT instruction for details. Format: U16

Format of the Notification Register



IP Register

IP Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1
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 **Generate State 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.

IP Register Workaround

Register and Subregister Numbers for IP Register

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

IP Register Fields

DWord	Bits	Subfield Description
0	31:3	**Ip**. Specifies the current instruction pointer. This pointer is relative to the **General State Base Address**.
	2:0	**Reserved**. MBZ.

TDR Registers

TDR Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1011b
Number of Registers:	1
Default Value:	No
Normal Access:	RO/CW
Elements:	8
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	No
Indexable?	No

There are 8 thread dependency registers (tdr0.0:uw to tdr0.7:uw) used by HW for the *sendc* instruction. These registers are read-only and can be accessed in 16-bit granularity.

When accessed explicitly, each thread dependency register has FFTID in the lower 8 bits, bits 8 to 14 are

forced to zero by HW. Bit 15 is the valid bit, which indicate whether the current thread has a dependency on the dependency thread stored in this thread dependency register.

The thread dependency registers are read only, the valids can only be set with a thread dispatch, and are reset by broadcasting end of thread messages after a thread retired. The FFTID's can only be changed with a thread dispatch. Any write into any of the TDR registers will clear the valid bit for the particular TDR if the write enable is true, the FFTID portion is strictly read only.

Register and Subregister Numbers for TDR Registers

RegNum[3:0]	SubRegNum[4:0]
1011b = tdr0 All other encodings are reserved.	00000b = **tdr0.0:uw** 00010b = **tdr0.1:uw** 00100b = **tdr0.2:uw** 00110b = **tdr0.3:uw** 01000b = **tdr0.4:uw** 01010b = **tdr0.5:uw** 01100b = **tdr0.6:uw** 01110b = **tdr0.7:uw** All other encodings are reserved.

TDR Registers Fields

DWord	Bits	Description
3	31	**Valid7**. This field indicates whether the thread specified by FFTID7 is still in-flight.
	30:25	**Reserved**. MBZ
	24:16	**FFTID7**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
	15	**Valid6**. This field indicates whether the thread specified by FFTID6 is still in-flight.
	14:9	**Reserved**. MBZ
	8:0	**FFTID6**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
2	31	**Valid5**. This field indicates whether the thread specified by FFTID5 is still in-flight.
	30:25	**Reserved**. MBZ

DWord	Bits	Description
	24:16	**FFTID5**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
	15	**Valid4**. This field indicates whether the thread specified by FFTID4 is still in-flight.
	14:9	**Reserved**. MBZ
	8:0	**FFTID4**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
1	31	**Valid3**. This field indicates whether the thread specified by FFTID3 is still in-flight.
	30:25	**Reserved**. MBZ
	24:16	**FFTID3**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
	15	**Valid2**. This field indicates whether the thread specified by FFTID2 is still in-flight.
	14:9	**Reserved**. MBZ
	8:0	**FFTID2**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
0	31	**Valid1**. This field indicates whether the thread specified by FFTID1 is still in-flight.
	30:25	**Reserved**. MBZ
	24:16	**FFTID1**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9
	15	**Valid0**. This field indicates whether the thread specified by FFTID0 is still in-flight.
	14:9	**Reserved**. MBZ
	8:0	**FFTID0**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U9

Performance Registers

Performance Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1100b
Number of Registers:	1
Default Value:	0h
Normal Access:	RO/RW
Elements:	4
Element Size:	32 bits
Element Type:	UD

Timestamp Register

This register is a low latency timestamp source, "TM", available as part of a thread's Architectural Register File (ARF). This is a is free running counter, 64b in size, and exposed to the ISA as individual 32b high 'TmHigh' and low 'TmLow' unsigned integer source operands. As part of the EU's register space, access to the timestamp has a low and deterministic latency and therefore can be used for intra-kernel high resolution performance profiling.

The TM features provides a 1-bit indicator 'TmEvent' which identifies the occurrence of a time-impacting event such as context switch or frequency change since the last time any portion of the Timestamp register value was read by that thread. Software that uses the Timestamp capability should check this bit to identify when a relative time calculation may be suspect. To properly use this additional information, the instrumentation code should operate on the Timestamp register value as a whole (i.e. as an 8 dword register) so that the 64b time and this 1b value are captured simultaneously, as opposed to 32b portions, to eliminate the chance of missing a TmEvent that might occur between accesses to 32b portions of this register.

Programming Note	
Context:	Performance Registers
The Timestamp register is saved as part of thread state on context-save, but only 'TmEvent' is restored (and technically always restored to '1' as a 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.

Register and Subregister Numbers for Performance Register

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

Performance Register Fields

DWord	Bits	Description
0 (tm0.0:ud)	31:0	**TmLow.** The lower 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32
1 tm0.1:ud	31:0	**TmHigh.** The upper 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32
2 tm0.2:ud	31:1	**Reserved**
	0	**TmEvent.** Indicates a discontinuous time-impacting event (e.g. context switch, frequency change) occurred since any portion of the Timestamp register was last read, thus making any relative duration calculation based on this counter suspect. This bit is reset at the time a new thread is loaded, and on each read of any portion of the 'Timestamp' register.
3 tm0.3 (pm0)	31:0	pm0. Increments based on the EU flexible performance event currently selected being true. Format: U32

Flow Control Registers

Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1101b
Number of Registers:	39
Default Value:	None
Normal Access:	RW*

Register and Subregister Numbers for Flow Conrol 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**. 00001b = **fc3.1**. 00010b = **fc3.2**. 00011b = **fc3.3**. All other encodings are reserved.
0100b = fc4	00000b = **fc4.0**. 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 behaviour 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 a GEN 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.

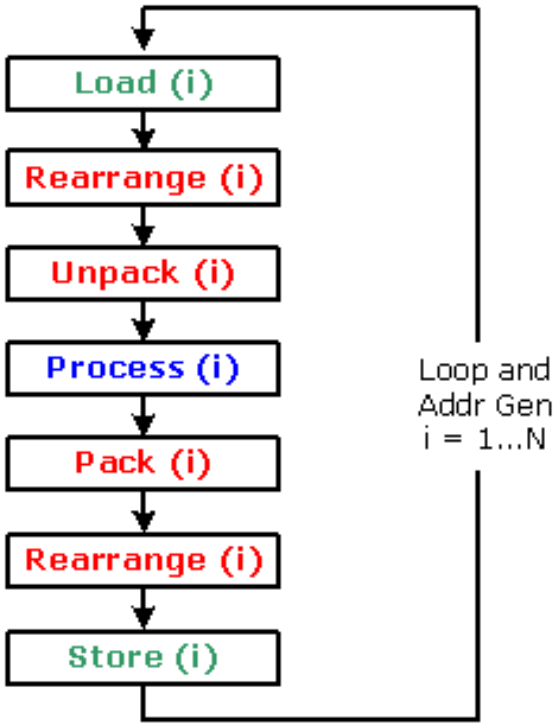
When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (***v***) and equivalent to a DWord for an immediate float vector (***vf***).

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 GEN 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 GEN. Another benefit is allowing high throughput of media applications, which tend to operate on byte or word data elements.

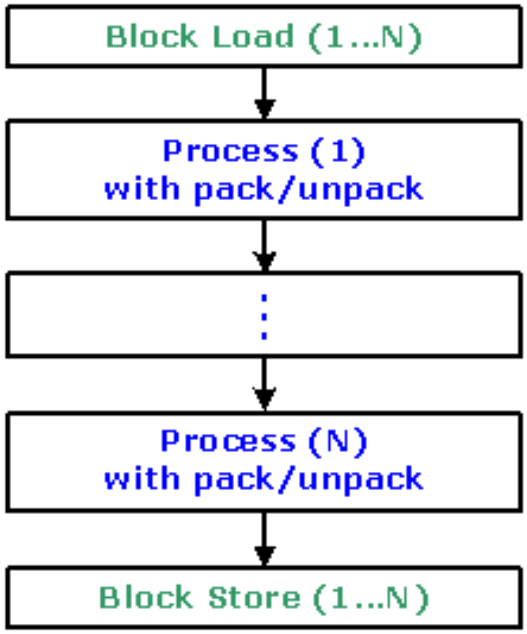
This can be illustrated by the example shown in *Region Parameters* and *Region Parameters*. As shown in *Region Parameters*, a sequence of SIMD instruction is executed on a conventional load/store based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first 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 GEN GRF register file, the outer loop may be unrolled for GEN. 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 GEN to combine data conditioning operations with computing operations. These techniques in GEN architecture help to achieve high compute efficiency and throughput for graphics and media applications.

Conventional SIMD Instruction Sequence



B6899-01

GEN SIMD Instruction Sequence for the Same Program



B6900-01

In a GEN 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.

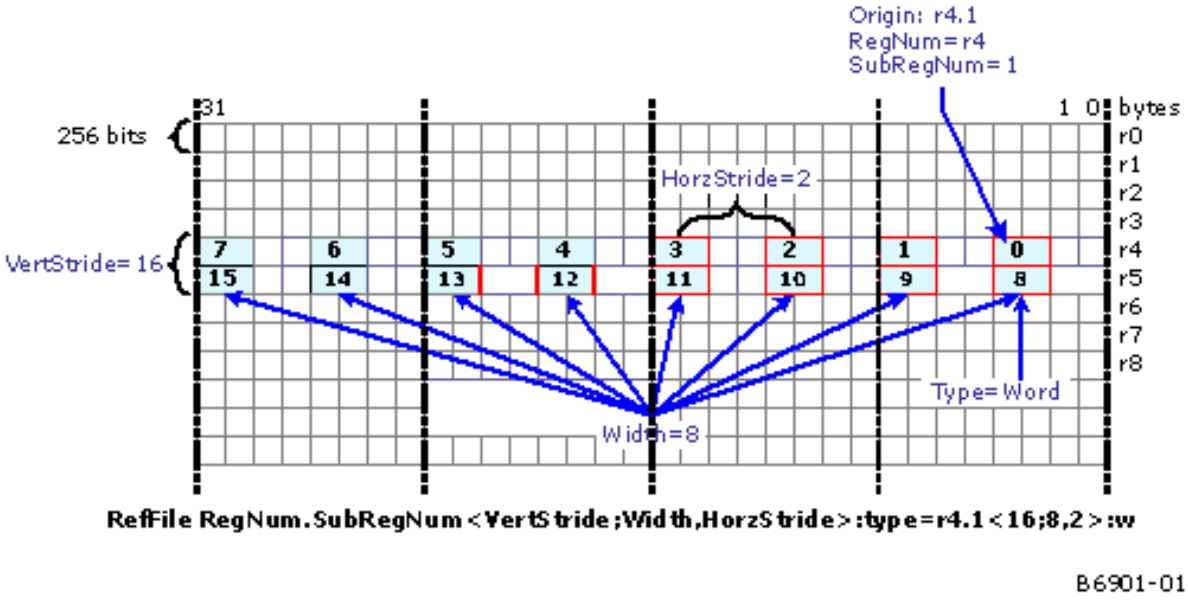
- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
    - Exception of this rule is present for the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

Description
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

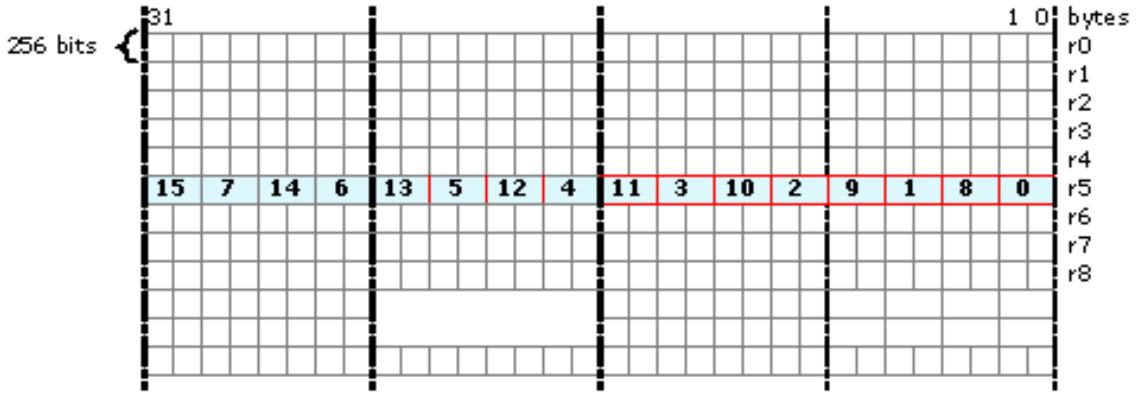


B6901-01

Region Parameters shows another example where the rows are interleaved. The region, having word data elements, starts at location r5.0:w. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location 5.2:w. And there are 8 elements per row. VertStride, the distance between two rows, is only 1 word, which is less than HorzStride. Therefore, the first element of the second row (channel number 8) is at r5.1:w, just next to channel number 0. It is clear from the picture that the two rows are interleaved.

By varying the region parameters, reader may construct other configurations. The next section provides more details on the region-based register addressing. However, there are restrictions imposed by hardware implementation, which can be found in the later sections of this chapter.

A 16-element register region with interleaved rows (*r5.0<1;8,2>:w*)



Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.

```
// Input: Type: ub | b | uw | w | ud | d | f | v
//RegNum: In unit of 256-bit register
//SubRegNum: In unit of data element size
//ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels
int ElementSize = (Type=="b"||Type=="ub") ? 1 : (Type=="w"|Type=="uw") ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum«5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
```

```
        int Offset = RowBase;
        for (int x=0; x<Width; x++) {
Address [Channel++] = Offset;
Offset += HorzStride*ElementSize;
        }
        RowBase += VertStride * ElementSize;
}
```

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 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

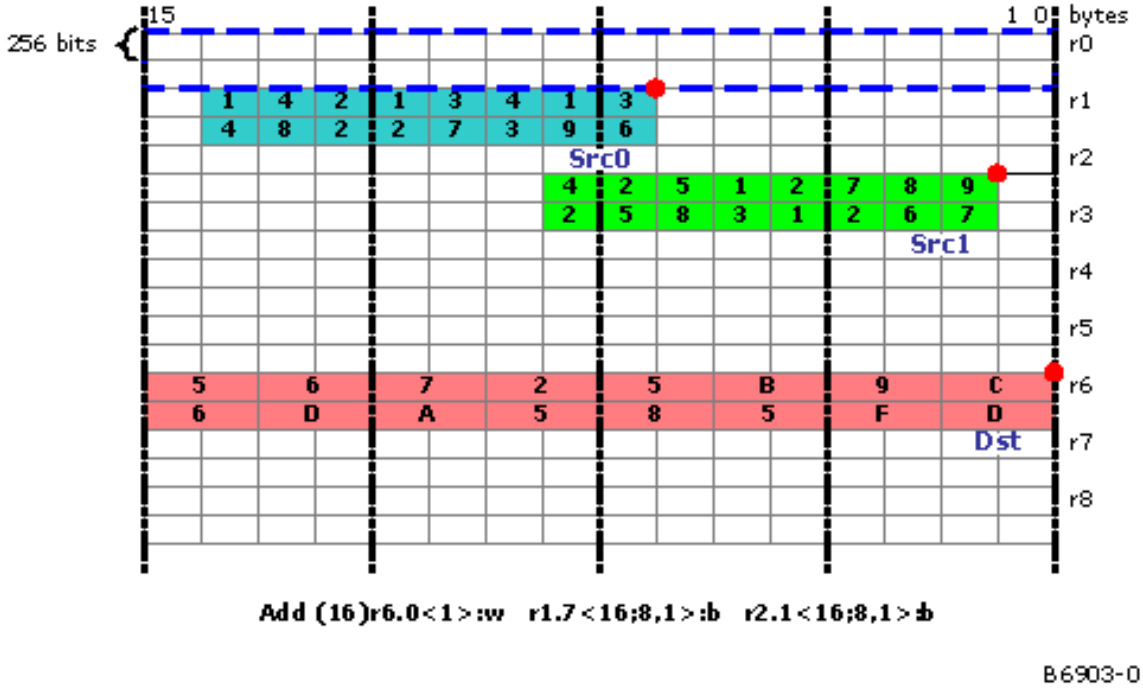
In this mode, all register region parameters are specified for an operand using fields in the instruction word.

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

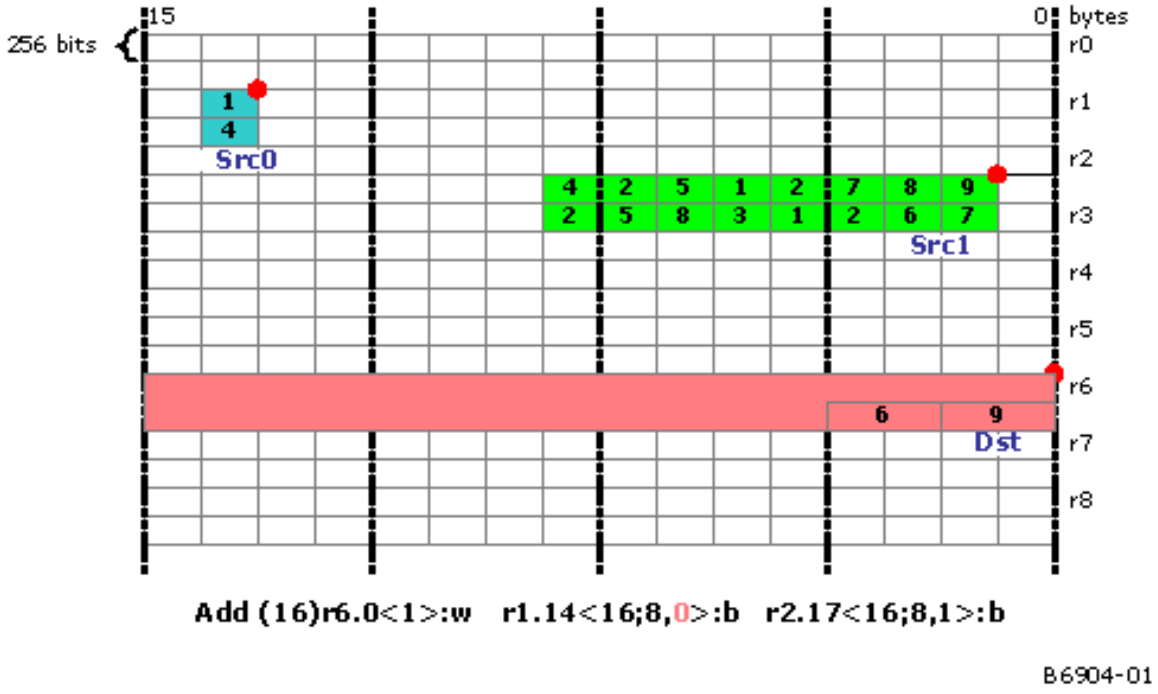
256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have C = 3+9.

A region description example in direct register addressing



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 r1.20 (20 = 14+16). Effectively, the 16 data elements of *Src0* are {1,1,1,1,1,1,1,1, 4,4,4,4,4,4,4,4}.

A region description example in direct register addressing with src0 as a vector of replicated scalars

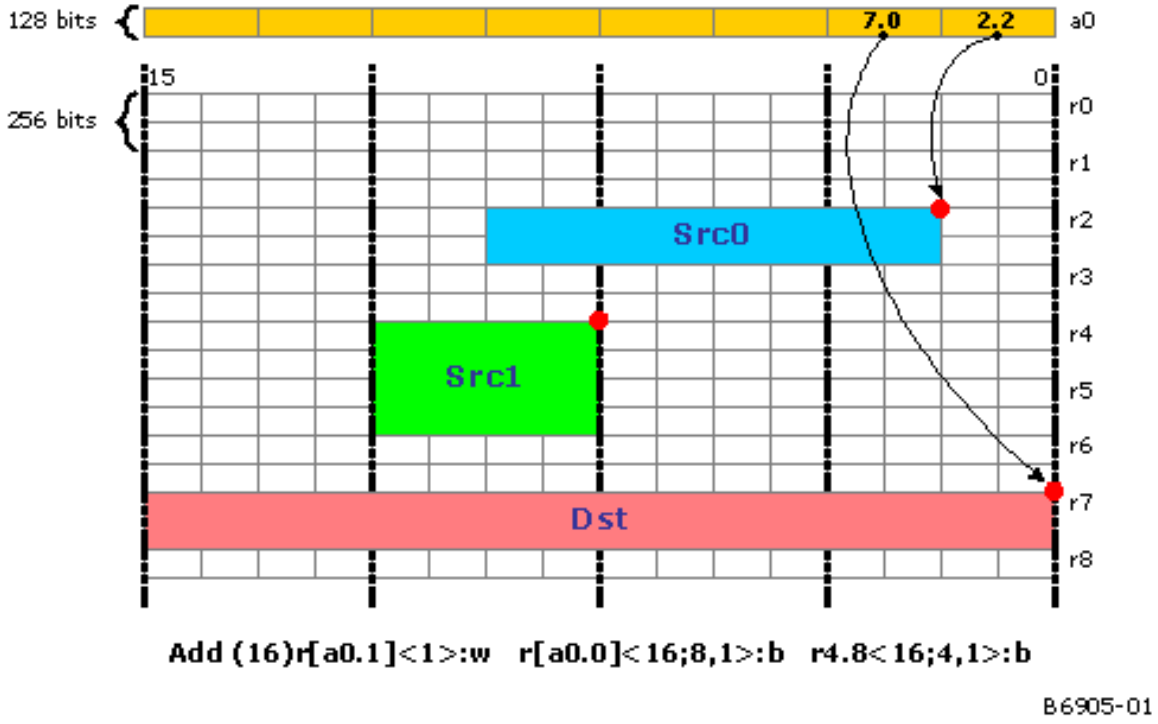


Register-Indirect Register Addressing with a 1x1 Index Region

In the register-indirect register addressing mode with 1x1 index region, the region origin is provided by the content of the address register, the rest of region parameters are provided by the fields in the instruction word.

Register-Indirect Register Addressing with a 1x1 Index Region depicts an example for this addressing mode. For example, the presence of a full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.

An example illustrating register-indirect register addressing mode with a 1x1 index region



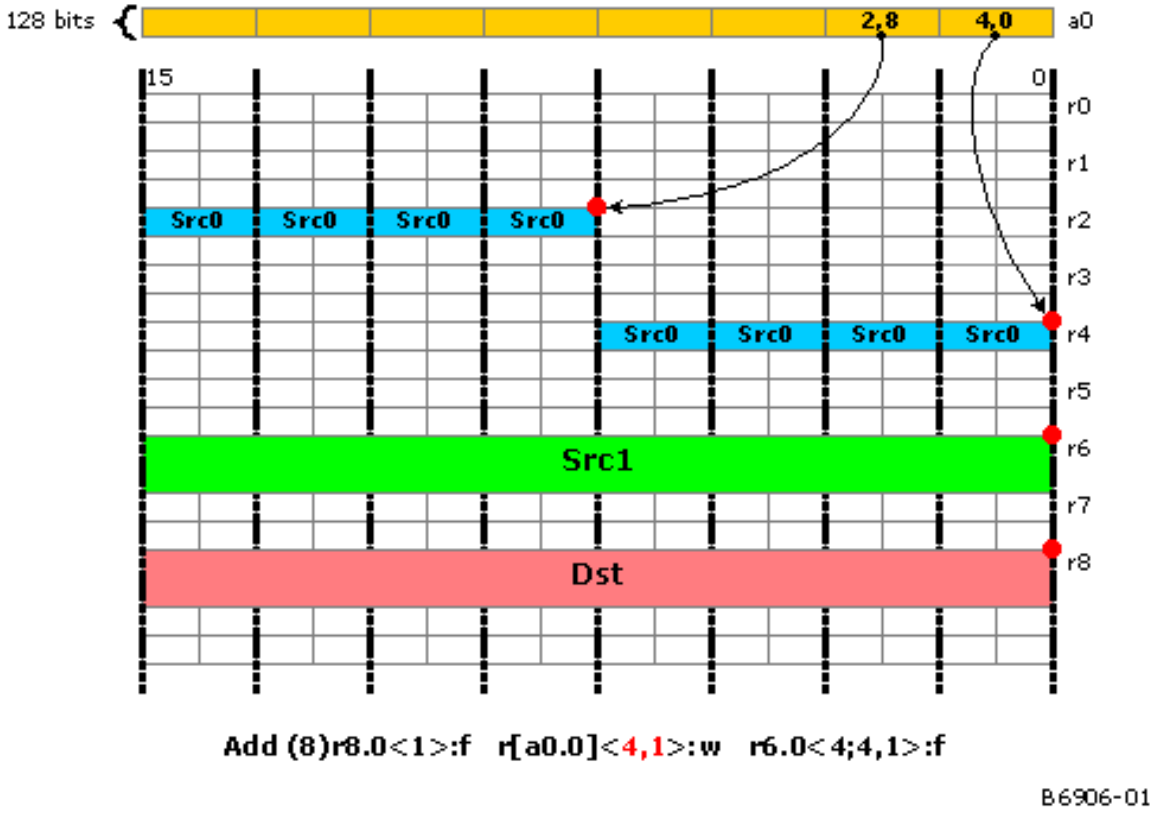
Register-Indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, the horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided and the following contiguous address registers are for the following rows. The total number of address registers used is inferred from the parameters *ExecSize* and *Width*.

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)

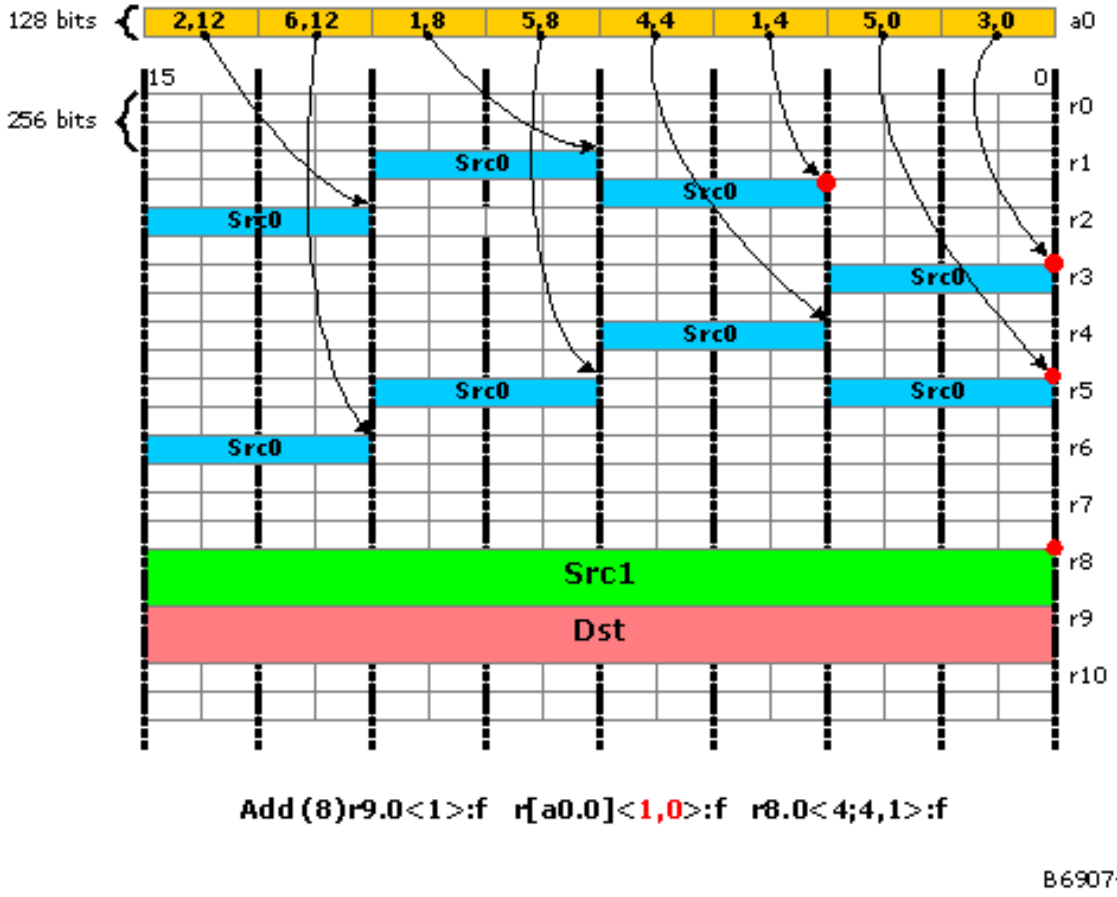


Register-Indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When *Width* of the region is 1, the number of address registers used equals *ExecSize*.

An example is provided in *Register-Indirect Register Addressing with a VxH Index Region*. The absent of vertical stride in the region description <1,0> with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As ExecSize/Width = 8, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.

An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



Access Modes

There are two basic GEN register access modes controlled by a single bit instruction subfield called Access Mode.

- 16-byte Aligned Access Mode (**align16**): In this mode, the origins of all operands (sources and destination), whether it is by direct addressing or register-indirect addressing, are 16-byte aligned. For example a row in the region description starts at 16-bype aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
    - The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.
- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region register descriptions are supported, however, source swizzle or destination mask are not supported.
    - The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.
-

Execution Data Type

The GEN architecture carries out arithmetic and logical operations using a smaller set of data types than the variety supported as source or destination operands. These are the *execution data types*. A particular arithmetic or logical instruction has one execution data type, from those listed in the table.

Execution Data Types

Type	Description
W	Word. 16-bit signed integer.
D	Doubleword. 32-bit signed integer.
Q	Quadword. 64-bit signed integer.
F	Float. 32-bit single precision floating-point number.
DF	Double Float. 64-bit double precision floating-point number.
HF	Half Float. 16-bit half precision floating-point number.

The following rules explain the conversion of multiple source operand types, possibly a mix of different types, to one common execution type:

- For floating-point sources, all source operands must have the same floating-point type, with the exceptions below:
  - A two-source floating-point instruction can have Float as the src0 type and VF (Packed Restricted Float Vector) as the immediate src1 type.
- Mixing floating-point and integer source types is not allowed. Either all source types must be one floating-point type or all source types must be integer types.
- Unsigned integers are converted to signed integers.
- Byte (B) or Unsigned Byte (UB) values are converted to a Word or wider integer execution type.
- If source operands have different integer widths, use the widest width specified to choose the signed integer execution type.

Note that when the execution data type is an integer type, it is always a signed integer type. For integer execution types, extra precision is provided within the hardware, including the accumulators, so that conversions from unsigned to signed do not affect instruction correctness.

Register Region Restrictions

A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist.

The following register region rules apply to the GEN implementation.

1. **General Restrictions Based on Operand Types**

   There are these general restrictions based on operand types:

   1. Where *n* is the largest element size in bytes for any source or destination operand type, *ExecSize* * *n* must be <= 64.
   2. When the Execution Data Type is wider than the destination data type, the destination must be aligned as required by the wider execution data type and specify a *HorzStride* equal to the ratio in sizes of the two data types. For example, a *mov* with a D source and B destination must use a 4-byte aligned destination and a *Dst.HorzStride* of 4.

2. **General Restrictions on Regioning Parameters**

   The mapping of data elements within the region of a source operand is in row-major order and is determined by the region description of the source operand, the destination operand, and the *ExecSize*, with these restrictions:

   1. *ExecSize* must be greater than or equal to *Width*.
   2. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.
   3. If *ExecSize* = *Width* and *HorzStride* = 0, there is no restriction on *VertStride*.
   4. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
   5. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be 0.
   6. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.
   7. *Dst.HorzStride* must not be 0.
   8. *VertStride* must be used to cross GRF register boundaries. This rule implies that elements within a '*Width*' cannot cross GRF boundaries.

3. **Region Alignment Rules for Direct Register Addressing**
   1. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.
   2. A destination cannot span more than 2 adjacent GRF registers.
   3. When a source or destination spans two registers, there are restrictions that vary by project, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

      ### Source or Destination Spans Two Registers

4. **Special Cases for Byte Operations**

1. When the destination type is byte (UB or B) only a 'raw move' using the *mov* instruction supports a packed byte destination register region: *Dst.HorzStride* = 1 and *Dst.DstType* = (UB or B). This packed byte destination register region is not allowed for any other instructions, including a 'raw move' using the *sel*instruction, because the *sel* instruction is based on Word or DWord wide execution channels.
2. There is a relaxed alignment rule for byte destinations. When the destination type is byte (UB or B), destination data types can be aligned to either the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case the destination data bytes can be either all in the even byte locations or all in the odd byte locations.

   This rule has two implications illustrated by this example:

```
// Example:
mov (8) r10.0<2>:b r11.0<8;8,1>:w
mov (8) r10.1<2>:b r11.0<8;8,1>:w

// Dst.HorzStride must be 2 in the above example so that the destination
// subregisters are aligned to the execution data type, which is :w.
// However, the offset may be .0 or .1.
// This special handling applies to byte destinations ONLY.
```

5. **Special Cases for Word Operations**

There are some special cases for word operations for specific projects, described in the following table. 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 Double Precision Data Types

Requirement

In Align16 mode, all regioning parameters must use the syntax of a pair of packed floats, including channel selects and channel enables.
`// 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.`

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.
`// Example:`
`mov (4) r10.0<1>:df r11.0<4;4,1>:df`
`// The above instruction moves four double floats.`
`// Note the difference between [DevIVB] and [DevHSW+].`

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
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.

For Align16 mixed mode, both input and output packed f16 data must be oword aligned, no oword crossing in packed f16.
Examples:
`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.`
 	
`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.`

In Align16 mode, replicate is supported and is coissueable. `mad(8) r20.xyzw:hf r3.0.r:f r6.0.xyzw:hf r6.0.xyzw:hf {Q1}`

Requirement

No SIMD16 in mixed mode when destination is packed f16 for both Align1 and Align16.
`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}`

No accumulator read access for Align16 mixed float.
When source is float or half float from accumulator register and destination is half float with a stride of 1, the source must register aligned. i.e., source must have offset zero.

No swizzle is allowed when an accumulator is used as an implicit source or an explicit source in an instruction. i.e. when destination is half float with an implicit accumulator source, destination stride needs to be 2.
`mac(8) r3<2>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf`
`mov(8) r3<1>:f acc0.0<8;4,2>:hf`

In Align16, vertical stride can never be zero for f16 `add(8) r3.xyzw:hf r4.0<4>xyzw:f r6.0<0>.xyzw:hf`

Math operations for mixed mode:
- In Align16, only packed format is supported
`math(8) r3.xyzw:hf r4.0.<4>xyzw:f r6.0<0>.xyzw:hf 0x09`
- In Align1, f16 inputs need to be strided
`math(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf`

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. `add(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf`
For mixed float operations, f16 datatype write to accumulator cannot be packed destination.

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`

Rules

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 a VxH addressing mode is used on src0, the destination may use one or two registers.
// Example:
// Case (a)
add (16) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud
add (8) r[a0.0, 8*4]<1>:d r[a0.2]<4,1>:ud r17.0<8;8,1>:ud
// Since the pointer (index register) is incremented every 4 elements
// (width), the second instruction moves from a0.0 to a0.2.
 	
// Case (b)
add (16) r10.0<2>:uw r[a0.0, 0]<1,0>:uw r16.0<8;8,1>:uw
// The above instruction behaves the same as the following two instructions:
add (8) r10.0<2>:uw r[a0.0, 0]<1,0>:uw r16.0<8;8,1>:uw
add (8) r11.0<2>:uw r[a0.8, 0]<1,0>:uw r17.0<8;8,1>:uw
// Since the pointer (index register) is incremented every 1 element
// (width), the second instruction moves from a0.0 to a0.8.
 	
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.

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.

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.
`// 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`

Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.
`// Example:`
`add (8) r10.0<2>:hf r11.0<8;8,1>:w r12.0<8;8,1>:w`
`// Destination stride must be 2.`
`mov (8) r10.0<2>:w r11.0<8;8,1>:hf`
`// Destination stride must be 2.`

The src, dst overlapping behavior with the second half src and the first half destination to the same register must not be used with any compressed instruction.

Destination Operand Description

Destination Region Parameters

Based on the above restrictions, a subset of register region parameters are sufficient to describe the destination operand:

- Destination Register Origin
  - Destination Register Number and Destination Subregister Number for direct register addressing mode
  - A Scalar Destination Register Index for register-indirect-register addressing mode
- Destination Register ‘Region’ – Note that destination register region does not have full region description parameters
  - Destination Horizontal Stride

SIMD Execution Control

This section of the BSpec discusses SIMD execution, both with and without predication. See the subtopics for more details.

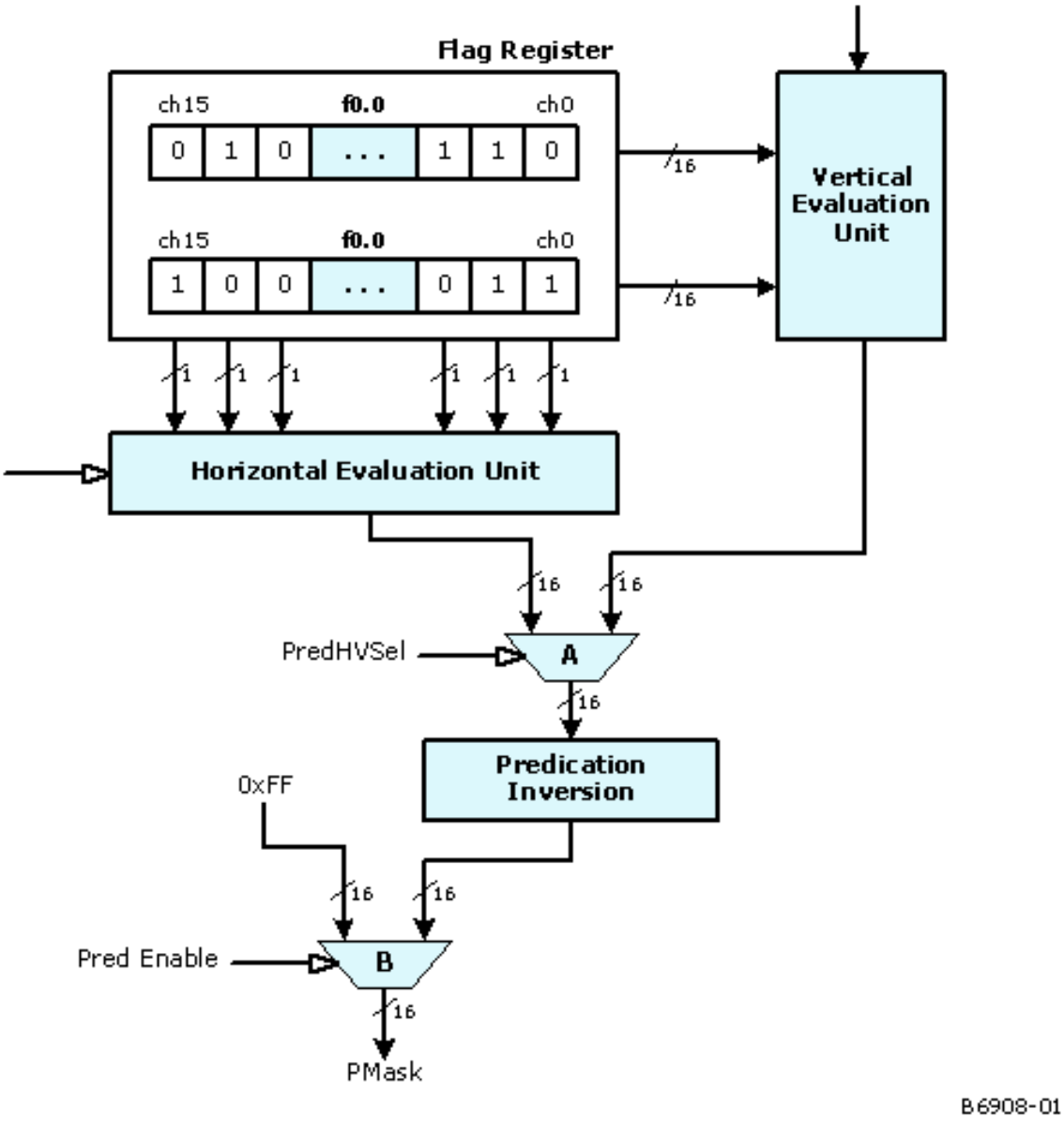
Predication

Predication is the conditional SIMD channel selection for execution on a per instruction basis. It is an efficient way of dynamic SIMD channel enabling without paying branch instruction overhead. When predication is enabled for an instruction, a Predicate Mask (PMask), which contains 16-bit channel enables, is generated internally in EU. Note that PMask is not a software visible register. It is provided here to explain how SIMD execution control works. PMask generation is based on the Predication Control (*PredCtrl*) field, Predication Inversion (*PredInv*) field and the flag source register in the instruction word. See Instruction Summary chapter for definition of these fields.

*Predication*shows the block diagram of the hardware logic to generate PMask. PMask is generated based on combinatory logic operation of the bits in the flag register. Instruction field *PredCtrl* controls the horizontal evaluation unit and vertical evaluation unit. MUX A in the figure selects whether horizontally-evaluated results or vertically-evaluated results are sent to the Predication Invertion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.

Generation of predication mask



No Predication

When PredCtrl field of a given instruction is set to 0 (“no predication”), it indicates that no predication is applied to this instruction. Effectively, the resulting PMask is all 1’s. This is shown by the 2:1 multiplexer B controlled by the Pred Enable signal in *Predication*. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.

Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is 'no combination' – the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```
If (PredCtrl == “Sequential flag channel mapping”) {
    For (ch=0; ch<16; ch++)
        PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}
```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of '.x', '.y', '.z' and '.w' select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = '.x' means that channel 0 in each group is replicated.
- OR combination: PredCtrl of '.any4h' means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of '.all4h' means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```
If (Access Mode == Align16) {
    For (ch = 0; ch < 16; ch += 4)
        Switch (PredCtrl) {
            Case     '.x':   bTmp = f0.#[ch]; break;
            Case     '.y':   bTmp = f0.#[ch+1]; break;
            Case     '.z':   bTmp = f0.#[ch+2]; break;
            Case     '.w':   bTmp = f0.#[ch+3]; break;
            Case '.any4h':   bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
Case '.all4h':   bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
        }
        bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
        PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
    }
}
```

In **Align1** access mode, horizontal combination is based on AND combination '.any#h' and OR combination '.all#h' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.

```
If (Access Mode == Align1) {
    Switch (PredCtrl) {
        Case  '.any2h':   groupSize = 2; <op> = '|'; break;         Case  '.all2h':
 groupSize = 2; <op> = '&'; break;
        Case  '.any4h':   groupSize = 4; <op> = '|'; break;         Case  '.all4h':
 groupSize = 4; <op> = '&'; break;
        Case  '.any8h':   groupSize = 8; <op> = '|'; break;         Case  '.all8h':
 groupSize = 8; <op> = '&'; break;
        Case '.any16h':   groupSize = 16; <op> = '|'; break;         Case '.all16h':
 groupSize = 16; <op> = '&'; break;     }
    For (ch = 0; ch < 16; ch += groupSize) {
        For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)
            bTmp = bTmp <op> f0.#[ch+inc];
        For (inc = 0; inc < groupSize; inc ++)
            PMask[ch+inc] = bTmp;
    }
}
```

Predication with Vertical Combination

Predication with vertical combination uses both flag subregister as inputs. The AND or OR combination is across the subregisters on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {
    For (ch = 0; ch < 16; ch ++) {
        If (PredCtrl == 'any2v')
            PMask[ch] = f0.0[ch] | f0.1[ch]
        Else If (PredCtrl == 'any2h')
            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 occured in any of the computation of the current instruction, including clamping, re-normalizing, and format conversion.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **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

GEN 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 GEN 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, GEN 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
Instructions other than send, may use this control as long as operations that have different pipeline latencies are not mixed. The operations that have longer latencies are: ▪ Opcodes *pln*, *lrp*, *dp**. ▪ Operations involving double precision computation. ▪ Integer DW multiplication where both source operands are DWs.

Non-present Operands

Some instructions do not have two source operands and one destination operand. If an operand is not present for an instruction the operand field in the binary instruction must be filed with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand src0. In this case, the second source operand src1 must be set to null and also with the same type as the first source operand src0. It is a special case when src0 is an immediate, as an immediate src0 uses DW3 of the instruction word, which is normally used by src1. In this case, src1 must be programmed with register file ARF and the same data type as src0.

Instruction Prefetch

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 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

This chapter contains these sections that introduce this volume.

- Introducing the Execution Unit
- EU Terms and Acronyms
- EU Changes by Processor Generation
- EU Notation

Subsequent chapters cover:

- EU Data Types
- Execution Environment
- Exceptions
- Instruction Set Summary
- Instruction Set Reference
- EU Programming Guide

The EU Programming Guide provides some useful examples and information but is not a complete or comprehensive programming guide.

Introducing the Execution Unit

This section introduces the Execution Unit (EU), a simple and capable processor within the GPU that supports graphics processing within the graphics pipelines, can do general purpose computing (GPGPU), and responds to exceptional conditions via the System Routine.

The EU provides parallelism at two levels: thread and data element. Multiple threads can execute on the EU; the number executing concurrently depends on the processor and is transparent to EU code. Each thread has its own registers (GRF and ARF, described below). Most EU instructions operate on arrays of data elements; the number of data elements is normally the *ExecSize (execution size)* or number of channels for the instruction. A *channel* is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels is independent of the number of physical ALUs or FPUs for a particular graphics processor.

EU native instructions are 128 bits (16 bytes) wide. Some combinations of instruction options can use compact instruction formats that are 64 bits (8 bytes) wide. Identifying instructions that can be compacted and creating the compact representations is done by software tools, including compilers and assemblers.

Data manipulation instructions have a destination operand (dst) and one, two, or three source operands (src0, src1, or src2). The instruction opcode determines the number of source operands. An instruction's last source operand can be an immediate value rather than a register.

Data read or written by a thread is generally in the thread's *GRF (General Register File)*, 128 general registers, each 32 bytes. A data element address within the GRF is denoted by a register number (r0 to r127) and a subregister number. In the instruction syntax, subregister numbers are in units of data element size. For example, a :d (Signed Doubleword Integer) element can be in subregister 0 to 7, corresponding to byte numbers in the instruction encoding of 0, 4, ... 28.

The EU cannot directly read or write data in system memory.

Specialized registers used to implement the ISA are in a distinct per thread *Architecture Register File (ARF)*. Each such register or group of related registers has its own distinct name. For example, ip is the instruction pointer and f0 is a flags register. An ARF register can be a src0 or dst operand but not a src1 or src2 operand. There are restrictions on how particular ARF registers are accessed that should be understood before directly reading or writing those registers. See the ARF Registers section for more information.

The EU supports both integer and floating-point data types, as described in the Numeric Data Types section.

For EU flow control, each channel has its own per-channel instruction pointer (PcIP[n]) and only executes an instruction when IP == PcIP[n] and any other masks enable the channel. Most flow control instructions use signed offsets from the current instruction address to reference their targets. Unconditional branches are done using *mov* with IP as the destination. Flow control can also use SPF (Single Program Flow) mode to execute with a single instruction pointer (IP).

The EU ISA supports predication, masking, regioning, swizzling, some type conversions, source modification, saturation, accumulator updates, and flag updates as part of instruction execution:

- *Predication* creates a bit mask (*PMask*) to enable or disable channels for a particular instruction execution. Pmask is derived from flag register and subregister values using boolean formulas determined by the PredCtrl (Predicate Control) and PredInv (Predicate Inversion) instruction fields. See the Predication section.
- *Masking* is the overall process of determining which channels execute for a given instruction based on five factors:
  - Number of channels (only channels in [0, *ExecSize* - 1] can execute)
  - Execution mask (*EMask*)
  - Whether the channel is on the instruction (if not in Single Program Flow mode and MaskCtrl is not NoMask)
  - Predicate mask (*PMask*)
  - In Align16 mode, any enabling of channels using the Dst.ChanEn instruction field (if MaskCtrl is not NoMask).
- *Regioning* specifies an array of data elements contained in one or two registers, with options for scattering, interleaving, or repeating data elements in registers using width and stride values, subject to significant constraints. Regioning also includes access mode (Align1 or Align16) and addressing mode (Direct or Indirect). See the Registers and Register Regions section.
- *Swizzling* allows small scale reordering of data elements within groups of four at the input using the modulo 4 channel names x, y, z, and w. For example, a swizzle of .wzyx with an *ExecSize* of 8 reads execution channels 0 to 7 from these input channels: 3, 2, 1, 0, 7, 6, 5, and 4. Swizzling is only available in the Align16 access mode, described in the Execution Environment chapter.
- *Type Conversions* do any needed conversion from source data type to execution data type and from execution data type to destination data type. See Execution Data Type for more information. Each instruction description indicates what combinations of data types are supported.
- *Source Modification* modifies a source operand just before doing the requested operation. For a numeric operation, the choices are:
  - No modification (normal).
  - - indicating negation.
  - (abs) indicating absolute value.
  - -(abs) indicating a forced negative value.

  Source modification logically occurs after any conversion from source data type to execution data type. Each instruction description indicates whether it supports source modification.
- *Saturation* clamps result values to the nearest value within a saturation range determined by the destination type. For a floating-point type, the saturation range is [0.0, 1.0]. For an integer type, the saturation range is the entire range for that type, for example [0, 65535] for the UW (Unsigned Word) type. Each instruction description indicates whether it supports saturation.
- *Accumulator Updates* optionally update the accumulator register or registers in the ARF with destination values as a side effect of instruction execution. The AccWrCtrl instruction field enables accumulator updates. The Accumulator Disable flag in control register 0 (cr0) can be used to

disable accumulator updates, regardless of AccWrCtrl values; for example, this flag may be used in the System Routine.

- *Flag Updates* optionally update a flags register and subregister (f0.0, f0.1, f1.0, or f1.1) with conditional flags based on the CondModifier (Condition Modifier) instruction field. For example, a CondModifier of **.nz** (not zero) assigns flag bits based on whether result elements are not zero (1) or zero (0). Each instruction description indicates whether it supports the Condition Modifier and any restrictions on the values supported.

The EU is not required to execute steps in its internal pipeline sequentially or in order, so long as it produces correct results.

The assembler syntax uses spaces between operands and encloses ExecSize and any predicate in parentheses. Instruction mnemonics, register names, conditional modifiers, predicate controls, and type designators use lowercase. Function names used with the math instruction are UPPERCASE.

```
( pred ) inst cmod sat ( exec_size ) dst src0 src1 { inst_opt, ... }
```

General register destination regions use the syntax r*m.n*<*HorzStride*>:type. General register directly addressed source regions use the syntax r*m.n*<*VertStride*;*Width*,*HorzStride*>:type. You need to understand more about register regioning to understand all of these terms.

The following example assembly language instruction adds two packed 16-element single-precision Float arrays in r4/r5 and r2/r3 writing results to r0/r1, only on those channels enabled by the predicate in f0.0 along with any other applicable masks.

```
(f0.0) add (16) r0.0<1>:f r2.0<8;8,1>:f r4.0<8;8,1>:f
```

EU Terms and Acronyms

This section provides three tables describing EU general terms and acronyms, EU data types, and EU selected ARF registers.

EU General Terms and Acronyms

Term	Description
ALT mode	A floating-point execution mode that maps +/- inf to +/- fmax, +/- denorm to +/-0, and NaN to +0 at the FPU inputs and never produces infinities, denormals, or NaN values as outputs. See IEEE mode.
ALU	Arithmetic Logic Unit. A functional block that performs integer arithmetic and logic operations, as distinct from instruction fetch and decode, floating-point operations (see FPU), or messaging.
AOS	Array Of Structures. Also see SOA.
ARF	Architecture Register File, a distinct register file containing registers used to implement specific ISA features. For example the Instruction Pointer and condition flags are in ARF registers. See GRF.
byte	An 8-bit value aligned on an 8-bit boundary and the basic unit of addressing. Bits within a byte are denoted 0 to 7 from LSB to MSB.

Term	Description
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 *gradual underflow*.
DWord	Doubleword. A 32-bit (4-byte) value aligned on a 32-bit (4-byte) boundary. Bits within a DWord are denoted 0 to 31 from LSB to MSB.
EOT	End of Thread. A flag set on a *send* or *sendc* instruction to terminate a thread's execution on the EU.
EU	Execution Unit. The single GPU unit described in this volume. This volume describes individual data parallel execution paths within a thread in the EU as *channels*. A few fields, like EUID, use EU to refer to a particular hardware resource used to implement the overall EU.
exception	An error or interrupt condition that arises during execution that may transfer control to the System Routine. Some exceptions can be disabled, preventing such transfers. As defined in this volume, some errors do not produce exceptions.
ExecSize	The number of execution channels for a particular instruction. Channels within that number are enabled or disabled by various masks.
floating-point	Numeric types that allow fractional values and often a wider range than integer types. The EU supports binary floating-point types including the single precision type and the double precision type defined by the IEEE 754 standard.
GEN	GEN is sometimes used to refer to Intel's mainstream GPU architecture integrated with recent CPU generations including SNB (Sandy Bridge) and IVB (Ivy Bridge).
GRF	General Register File, a distinct register file containing 128 general registers, r0 to r127. Each general register is 256 bits (32 bytes), can contain any type of data, and can be accessed with any valid combination of addressing mode, access mode, and region parameters. A general register is directly addressed using a register number and subregister number, or indirectly addressed using an address subregister (index register) and an address immediate offset.
IEEE mode	A floating-point execution mode that supports all the kinds of floating-point values described by the IEEE 754 standard: normalized finite nonzero binary floating-point numbers, signed zeros, signed infinities, signed denormals that are closer to zero than any normalized value but still nonzero, and NaN (not a number) values. See ALT mode.
index register	An address subregister when used for indirect addressing.
inf	Infinity, +inf or -inf, as a floating-point value in IEEE mode.
instruction	In this volume, *instruction* always refers to an EU instruction.

Term	Description
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 GEN 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	A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist. The immediate vector data types are all described as *Packed* because each such type packs several small data elements into a 32-bit immediate value.
QWord	Quadword. A 64-bit (8-byte) value aligned on a 64-bit (8-byte) boundary. Bits within a QWord are denoted 0 to 63 from LSB to MSB.
region	A collection of data locations in registers and subregisters for a source or destination operand. The associated regioning parameters allow regions to be arrays with various layouts.
register	Part of the directly accessible state of an EU program, such as a general register in the GRF or an architecture register in the ARF. Note that system memory is not directly accessible.
SIMD	Single Instruction Multiple Data. Each EU instruction can operate on multiple data elements in parallel, as specified by the instruction's ExecSize.
SIP	System Instruction Pointer, the starting IP value for the System Routine.
SOA	Structure of Arrays. Also see AOS.
SPF	Single Program Flow. A mode in which every execution channel uses the common instruction pointer, IP in the ip register. The SPF bit in the control register is 1 to enable SPF and 0 to disable it. If SPF is disabled, then each execution channel n has its own instruction pointer, PcIP[n] and each channel n is only eligible to execute, subject to other masking, when PcIP[n] == IP.

Term	Description
swizzle	Rearrange data elements within a vector. The EU supports modulo four swizzling of register source operands at the input in the Align16 access mode.
System Routine	A global EU exception handling routine. Any enabled exception from any EU thread transfers control to this routine.
thread	An instance of a program executing on the EU. The life cycle for a thread on the EU starts with the first instruction after being dispatched to the EU by the Thread Dispatcher and ends after executing a *send* or *sendc* instruction with EOT set, signaling thread termination. Threads can be independent or can communicate with each other via the Message Gateway shared function.
word	A 16-bit (2-byte) value aligned on a 16-bit (2-byte) boundary. Bits within a word are denoted 0 to 15 from LSB to MSB. *Word* has denoted a 16-bit unit for Intel processors since the 8086 and 8088 processors were introduced in 1978.

The next table lists all EU numeric data types. See the Numeric Data Types section for more information about each data type.

EU Numeric Data Types (Listed Alphabetically by Short Name)

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
B	:b	Signed Byte Integer	1	8	I	Signed integer in the range -128 to 127.
D	:d	Signed Doubleword Integer	4	32	I	Signed integer in the range -2^{31} to $2^{31} - 1$.
DF	:df	Double Float	8	64	F	Double precision floating-point number.
F	:f	Float	4	32	F	Single precision floating-point number.
HF	:hf	Half Float	2	16	F	Half precision floating-point number.
Q	:q	Signed Quadword Integer	8	64	I	Signed integer in the range -2^{63} to $2^{63} - 1$.
UB	:ub	Unsigned Byte Integer	1	8	I	Unsigned integer in the range 0 to 255.
UD	:ud	Unsigned Doubleword Integer	4	32	I	Unsigned integer in the range 0 to $2^{32} - 1$.
UQ	:uq	Unsigned Quadword Integer	8	64	I	Unsigned integer in the range 0 to $2^{64} - 1$.
UV	:uv	Packed Unsigned Half Byte Integer Vector	4	32	I	Eight 4-bit unsigned integer values each in the range 0 to 15. Only used as an immediate value.
UW	:uw	Unsigned Word Integer	2	16	I	Unsigned integer in the range 0 to 65,535.
V	:v	Packed Signed Half Byte Integer Vector	4	32	I	Eight 4-bit signed integer values each in the range -8 to 7. Only used as an immediate value.
VF	:vf	Packed Restricted Float	4	32	F	Four 8-bit restricted float values. Only used

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
		Vector				as an immediate value.
W	:w	Signed Word Integer	2	16	I	Signed integer in the range -32,768 to 32,767.

The next table lists the seven ARF registers that you should understand first, omitting several others. See the ARF Registers section for more information, including descriptions of additional registers not listed below.

EU Selected ARF Registers (Listed Alphabetically by Name)

Name	Assembler Syntax	Description
Accumulators	acc0, acc1	Data registers that can hold integer or floating-point values of various sizes. Many instructions can implicitly update accumulators with a copy of destination values, done by setting the AccWrCtrl instruction option. A few instructions, like *mac* (Multiply Accumulate), use the accumulators as an implicit source operand, useful for some iterative calculations. There are added accumulator registers acc2 to acc9 for special purposes; these added accumulators are not generally used.
Address Register	a0.*s*	Holds subregisters primarily used for indirect addressing. Each subregister is a 16-bit UW (Unsigned Word) value. For an indirectly addressed operand or element, the subregister value plus an AddrImm signed offset field determines the byte address (RegNum and SubRegNum) within the register file (GRF). There are 16 address subregisters.
Control Register	cr0.*s*	Contains bit fields for floating-point modes, flow control modes, and exception enable/disable. Also contains exception indicator flags and saves the AIP (Application Instruction Pointer) on transferring control to the System Routine to handle an exception.
Flags	f*r.s*	Used as the outputs for various channel conditional signals, such as equality/zero or overflow. Used as the inputs for predication. There are two 32-bit flags registers each containing two 16-bit subregisters.
Instruction Pointer (IP)	ip	References the current instruction in memory, as an unsigned offset from the General State Base Address. IP is the thread's overall instruction pointer. Each channel n can have its own instruction pointer (PcIP[n]). If not in Single Program Flow mode (SPF is 0) then only those channels where PcIP[n] == IP are eligible to execute the instruction, if enabled by all other applicable masks.
Null Register	null	Indicates a non-existent operand. Unused operands in the instruction format, like the

Name	Assembler Syntax	Description
		unused second source operand field in a *mov* instruction, are encoded as null. For present source operands, reading a null source operand returns undefined values. For null destination operands, results are discarded but any implicit updates to accumulators or flags still occur.
State Register	sr0.*s*	Contains thread identification and scheduling fields, and mask fields for enabling or disabling channels.

Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on. Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple EU cores each with multiple threads can generally be ignored by software. There are some exceptions to this rule: e.g., for:

Description
thread-to-thread communication (see *Message Gateway*, *Media*)
synchronization of thread output to memory buffers (see *Geometry Shader*)

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs.

EU Changes by Processor Generation

This section describes how the EU changes for particular processor generations. Instruction compaction tables can differ for each generation, so that is not mentioned in these lists. Particular readers and audiences can see only certain content in this section. Workarounds for particular generations, SKUs, or steppings are not included in these lists. Some small changes in instruction layouts are not included in these lists.

Description
These features or behaviors are added for IVB+, continuing to later generations: • The maximum *ExecSize* increases to 32, for byte or word operands. • Increase the number of flag registers from one to two. • Add the NibCtrl field, used with QtrCtrl to select groups of channels or flags. • Add the DF (Double Float) data type, the first time an 8-byte data type is supported. DF only supports the IEEE floating-point mode and not the ALT floating-point mode. • Add a shared source data type field and a destination data type field for instructions with three source operands, allowing F (Float), DF (Double Float), D (Signed Doubleword Integer), or UD (Unsigned Doubleword Integer) types to be specified. • Add bit manipulation instructions: *bfi1*, *bfi2*, *bfrev*, *cbit*, *fbh*, and *fbl*. • Add the integer *addc* (Add with Carry) and *subb* (Subtract with Borrow) instructions. • Add the *brc* (Branch Converging) and *brd* (Branch Diverging) instructions. • For the *cmp* and *cmpn* instructions, relax the accumulator restrictions. • For the *sel* instruction, remove the accumulator restriction. • Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0.
These features or behaviors are added for HSW+, continuing to later generations: • DF (Double Float) operands use an element size of 8. Regioning and channel parameters for the DF type are determined normally, in the same way as for other types. • Add the channel enable register, flow control registers, and stack pointer register in the ARF. • In the Control Register, add the Force Exception Status and Control, Context Save Status, and Context Restore Status bits. • Relative instruction offsets (JIP, UIP) are now 32-bit values in units of bytes (rather than 16-bit values using 8-byte units) for some instructions: *brc*, *brd*, *call*, and *jmpi*. • A *call* instruction can get the relative instruction offset (JIP) from a register. • Add the *calla* (Call Absolute) instruction. • A *mov* instruction with different source and destination types can now use conditional modifiers.
These features or behaviors are added for BDW+, continuing to later generations: • Add the HF (Half Float) type and a corresponding HF execution data type and execution path. • Add flags to indicate IEEE floating-point exceptions and to enable or disable exception reporting to those flags. • Add the Single Precision Denorm Mode bit in Control Register 0. It can be enabled to allow calculations using the F (Float) type in IEEE floating-point mode to support denormals and gradual underflow. • Add the Q (Signed Quadword Integer) and UQ (Unsigned Quadword Integer) types. Integer source types cannot mix 64-bit and 8-bit operands. Some integer instructions (e.g., *avg*) do not support Q or UQ source types. • Instructions with one source operand and a 64-bit source type can have immediate 64-bit source operands.

Description
• The JIP and UIP relative instruction offset fields in all remaining flow control instructions (those instructions that did not make this change for HSW+) 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 SNB *fork* instruction. • Add the *join* instruction. • For the *lzd* instruction, remove the accumulator restriction. • The *mach* instruction reverses the roles of the two source operands compared to previous generations. • Add the *madm* instruction. • Enhance the *math* instruction to allow some immediate source values and support the INVM and RSQRTM functions. • For the *mul* instruction, relax the accumulator restriction on source operands so it applies for only integer source operands. • For the rounding instructions (*rndd*, *rnde*, *rndu*, and *rndz*), remove the accumulator restrictions. • Revise the *shl* and *shr* instructions to use the low 6 bits of the shift count in QWord mode, versus the low 5 bits otherwise. • Add the *smov* instruction. • Add eight accumulator registers, acc2 to acc9, used only for the special purpose of emulating IEEE-compliant fdiv and sqrt operations. • Add message control registers. • Widen the sp (Stack Pointer) register to 64 bits. • Add the IEEE Exception, Page Fault Status, and Page Fault Code bit fields in the State Register. • Remove some regioning restrictions when operands span two registers.
These features or behaviors are added for SKL+, continuing to later generations: • 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.

EU Notation

The Courier New font is used for code examples and for the Syntax, Format, and Pseudocode sections in the instruction reference.

The *italic* font style is used for instruction mnemonics outside of code (e.g., the *send* instruction), for syntactic production names, for key values in algorithms (*ExecSize*), and to emphasize a word or phrase. For example: When bit 10 is set, the destination register scoreboard is *not* cleared.

The **bold** font weight is used for the short name and long name of a bit field being described, for value names being defined, for syntactic terminals, for unnumbered subheadings, and for the terms Note or Workaround used to introduce a paragraph.

Bit field names and value names used where not being defined and not as syntactic terminals are in plain text.

Bit field values in hex use the 0x prefix. The BSpec currently uses the 0x prefix for hex in some parts and the h suffix for hex in other parts. For single bits, values appear as simply 0 or 1. For multi-bit binary values, the appropriate number of binary digits appears with a b suffix.

Instruction mnemonics are lowercase. Function names invoked using the *math* instruction are UPPERCASE. For example, SQRT.

Device names in the new syntax do not use the Dev prefix or square brackets and often appear in tagged tables with a blue background. For example:

Device names in the old syntax are in plain text in square brackets. For example, .

Tables describing bit field layouts or registers proceed from most significant to least significant bits. Figures showing bit fields or registers show most significant bits on the left and least significant bits on the right.

Any bit, field, or register described as Reserved should be regarded as undefined and unpredictable. Such bits should be treated as follows:

- When testing values, do not depend on the state of reserved bits. Mask out or otherwise ignore such bits.
- Sometimes software must initialize reserved bits. For example, a compiler must write complete instruction values when creating an instruction stream, including reserved bits. In such cases, write reserved bits as zeros unless otherwise indicated.
- Do not use reserved bits as extra storage for software-defined values; put nothing in such bits.
- When saving state and restoring state, save and restore any reserved bits as well.
- Do not assume that reserved bits are invariant between explicit writes. Software should function even if reserved bits change in undefined and unpredictable ways.

Any value, encoding, or combination of values or encodings described as Reserved must not be used. The EU's behavior is undefined in this case.

When a combination of instruction parameters or an EU state is described as producing undefined results or behavior, do not assume that undefined results or behavior are confined to specific instructions, operands, registers, or channels.

Execution Environment

EU Data Types

Fundamental Data Types

Numeric Data Types

Floating Point Modes

- IEEE Floating Point Mode
  - Partial Listing of Honored IEEE 754 Rules
  - Complete Listing of Deviations or Additional Requirements vs IEEE 754
  - Min/Max of Floating Point Numbers
- Alternative Floating Point Mode

Floating-Point Support

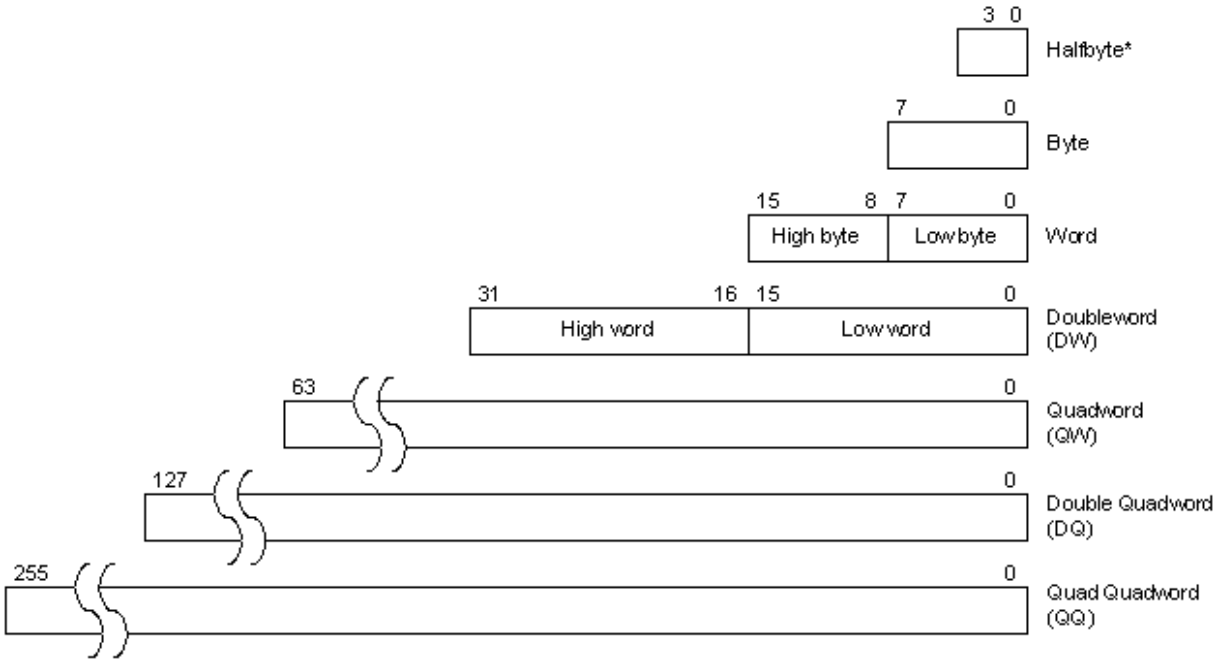
- IEEE Floating-Point Exceptions
- Floating-Point Compare Operations

Type Conversion

Fundamental Data Types

The fundamental data types in the GEN 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 GEN 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.

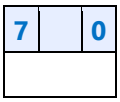
Numeric Data Types

The numeric data types defined in the GEN 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

7	6		0
S			

UW: Unsigned Word, 16-bit Unsigned Integer

15		0

W: Word, 16-bit Signed Integer

15	14		0
S			

UD: Unsigned Doubleword, 32-bit Unsigned Integer

31		0

D: Doubleword, 32-bit Signed Integer

31	30		0
S			

UQ: Unsigned Quadword, 64-bit Unsigned Integer BDW+

63		0

Q: Quadword, 64-bit Signed Integer BDW+

63	62		0
S			

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

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0

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

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
S		S		S		S		S		S		S		S	

The following table summarizes the EU integer data types.

Execution Unit Integer Data Types

Notation	Size in Bits	Name	Range
UB	8	Unsigned Byte Integer	[0, 255]
B	8	Signed Byte Integer	[-128, 127]
UW	16	Unsigned Word Integer	[0, 65535]
W	16	Signed Word Integer	[-32768, 32767]
UD	32	Unsigned Doubleword Integer	$[0, 2^{32} - 1]$
D	32	Signed Doubleword Integer	$[-2^{31}, 2^{31} - 1]$
UQ	64	Unsigned Quadword Integer	$[0, 2^{64} - 1]$
Q	64	Signed Quadword Integer	$[-2^{63}, 2^{63} - 1]$
UV	32	Packed Unsigned Half-Byte Integer Vector	[0, 15] in each of eight 4-bit immediate vector elements.
V	32	Packed Signed Half-Byte Integer Vector	[-8, 7] in each of eight 4-bit immediate vector elements.

Restriction: Only a raw move using the *mov* instruction supports a packed byte destination register region. For information about raw moves, refer to the **Description** in *mov – Move*.

Floating-Point Numeric Data Types

The Execution Unit supports the following floating-point data types.

Type or Description
The Float type uses the single precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
The Double Float type uses the double precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
In the ALT floating-point mode, representations for infinities, denorms, and NaNs within those formats are not used.

Type or Description
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 BDW+

15	14		10	9		0
S	biased exp.			fraction		

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

31	30		23	22		0
S	biased exponent			fraction		

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

63	62		52	51		0
S	biased exponent			fraction		

VF: Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number

31	30		28	27		24	23	22		20	19		16	15	14		12	11		8	7	6		4	3		0
S	b. exp.			frac.			S	b. exp.			frac.			S	b. exp.			frac.			S	b. exp.			frac.		

The following table summarizes the EU floating-point data types.

Execution Unit Floating-Point Data Types

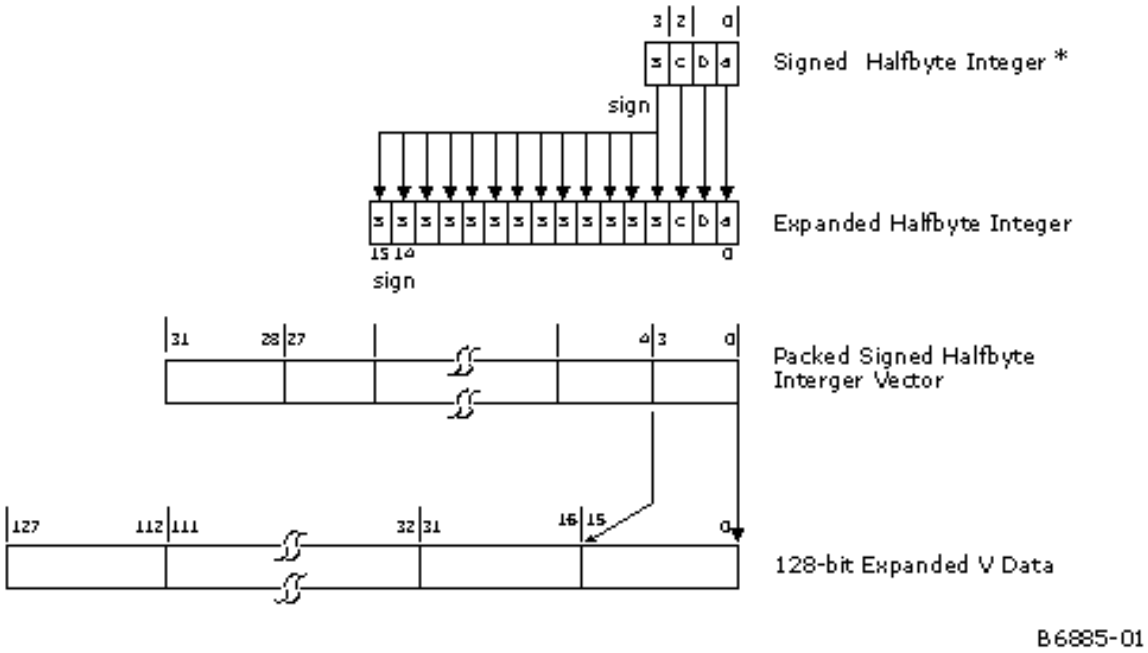
Notation	Size in Bits	Name	Range
HF	16	Half Float	Half precision, 1 sign bit, 5 bits for the biased exponent, and 10 bits for the significand: $[-(2-2^{-10})^{31}...-2^{-40}, 0.0, 2^{-40}... (2-2^{-10})^{31}]$
F	32	Float	Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand: $[-(2-2^{-23})^{127}...-2^{-149}, 0.0, 2^{-149}... (2-2^{-23})^{127}]$
DF	64	Double Float	Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: $[-(2-2^{-52})^{1023}...-2^{-1074}, 0.0, 2^{-1074}... (2-2^{-52})^{1023}]$
VF	32	Packed Restricted Float Vector	Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit, 3 bits for the biased exponent (bias of 3), and 4 bits for the significand: [–31... –0.1328125, -0, 0, 0.1328125 ... 31]

Packed Signed Half-Byte Integer Vector

A packed signed 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 a GEN instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN hardware converts the vector into an 8-element signed word vector by sign extension. This is illustrated in *Numeric Data Types*.

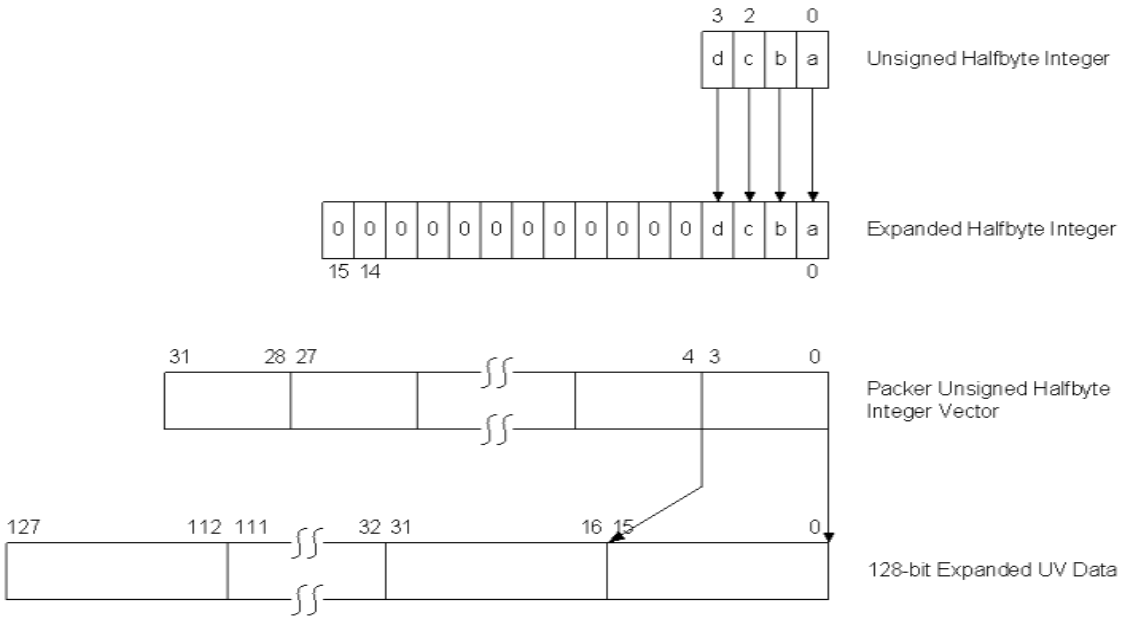
The short hand format notation for a packed signed half-byte vector is **V**.

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



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 a GEN instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN hardware converts the vector into an 8-element signed word vector.

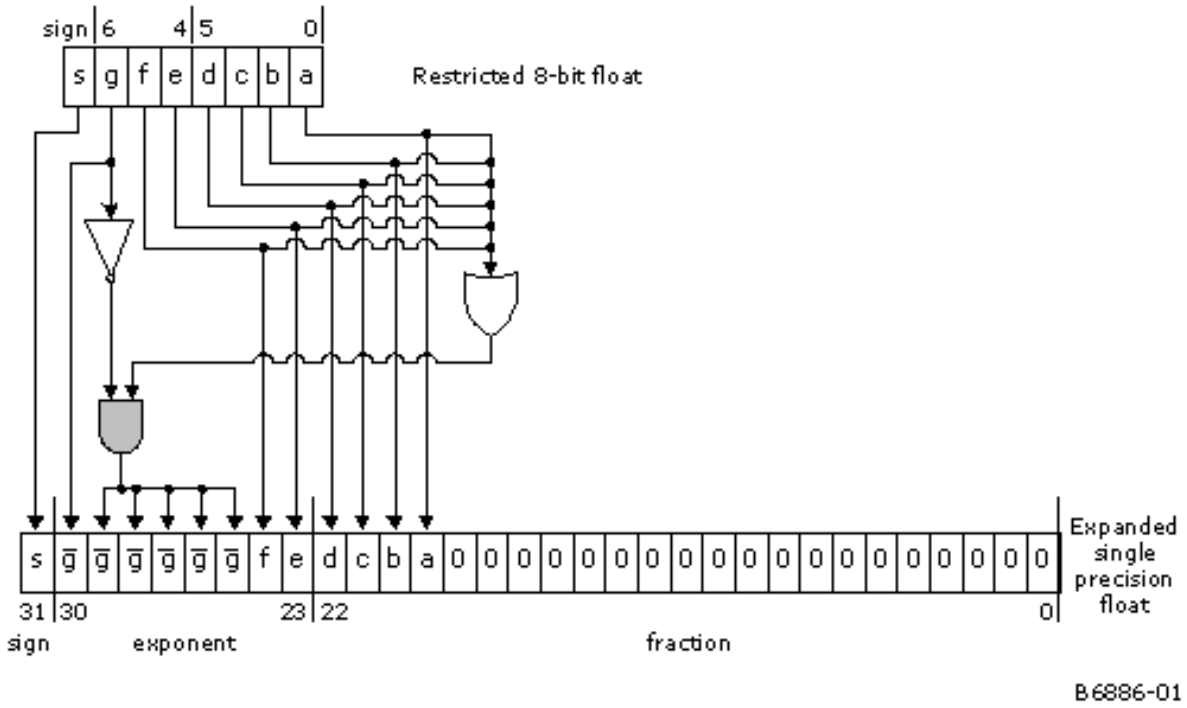


Packed Restricted Float Vector

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 a GEN instruction. It cannot be used for the destination operand, or a non-immediate source operand.

The following figure shows how to convert an 8-bit restricted float into a single precision float. Converting a 3-bit exponent with a bias of 3 to an 8-bit exponent with a bias of 127 is by adding 4, or equivalently copying bit 2 to bit 7 and putting the inverted bit 2 to bits 6:2. A special logic is also needed to take care of positive/negative zeros.

Conversion from a Restricted 8-bit Float to a Single-Precision Float



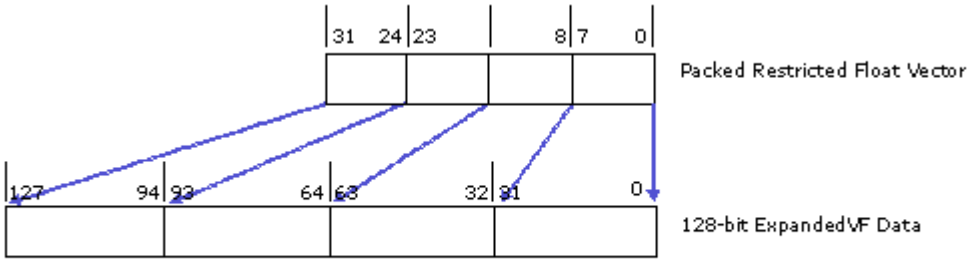
The following table shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinites, NaN, and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range – the magnitude of the smallest non-zero number is 0.1328125.

Examples of Restricted 8-bit Float Numbers

Class	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]	Extended 8-bit Exponent	Floating Number in Decimal
Positive Normalized Float	0x70-0x7F	0	111	0000 ... 1111	1000 0011	16 ... 31
	0x60-0x6F	0	110	0000 ... 1111	1000 0010	8 ... 15.5
	0x50-0x5F	0	101	0000 ... 1111	1000 0001	4 ... 7.75
	0x40-0x4F	0	100	0000 ... 1111	1000 0000	2 ... 3.875
	0x30-0x3F	0	011	0000 ... 1111	0111 1111	1 ... 1.9375
	0x20-0x2F	0	010	0000 ... 1111	0111 1110	0.5 ... 0.96875
	0x10-0x1F	0	001	0000 ... 1111	0111 1101	0.25 ... 0.484375
	0x01-0x0F	0	000	0001 ... 1111	0111 1100	0.1328125 ... 0.2421875
	0x00	0	000	0000	0000 0000	0 (+zero)
Negative Normalized Float	0xF0-0xFF	1	111	0000 ... 1111	1000 0011	-16 ... -31
	0xE0-0xEF	1	110	0000 ... 1111	1000 0010	-8 ... -15.5
	0xD0-0xDF	1	101	0000 ... 1111	1000 0001	-4 ... -7.75
	0xC0-0xCF	1	100	0000 ... 1111	1000 0000	-2 ... -3.875
	0xB0-0xBF	1	011	0000 ... 1111	0111 1111	-1 ... -1.9375
	0xA0-0xAF	1	010	0000 ... 1111	0111 1110	-0.5 ... -0.96875
	0x90-0x9F	1	001	0000 ... 1111	0111 1101	-0.25 ... -0.484375
	0x81-0x8F	1	000	0001 ... 1111	0111 1100	-0.1328125 ... -0.2421875
	0x80	1	000	0000	0000 0000	-0 (-zero)

The following figure shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The shorthand format notation for a packed signed half-byte vector is VF.



Floating Point Modes

GEN 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 GEN execution units, GEN 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 GEN 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.
Source modifiers are not applied to NAN.
Handle denorms as follows: • When inputs are denorms in mixed mode (one of the source operand is half float and other is single precision float OR source is half float and destination is single precision float), on upconversion, they are non-denorms. So computes in 32b can handle this. • When outputs are denorms, the denorms are retained if there is a format conversion. • 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:

Description
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 GEN 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.

Description
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.

Alternative Floating Point Mode 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

Alternative Floating Point Mode 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 GEN 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
I Rcp (and rsq) of INF yields +/- 0	(+/-)INF is never present as input
(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0	There is no divide in Sampler

Floating-Point Support

The following sections describe BDW+ 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, BDW+ 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.

Description
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 for BDW 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 <= 0.5 *ulp* (unit in the last place) when using the round to nearest or even rounding mode:

ADD (*add* instruction)

DIV (*math* instruction with FDIV function code)

FMA (fused multiply add, *mad* instruction)

MUL (*mul* instruction)

SQRT (*math* instruction with SQRT function code)

SUB (*add* instruction using one **-** source modifier)

Conversions (float to float, float to int, and int to float)

Min/Max

Compare

Single Precision Floating-Point Rounding to Integral Values

The *rndd* (Round Down), *rnde* (Round to Nearest or Even), *rndu* (Round Up), and the *rndz* (Round to Zero) instructions round arbitary Float values to integral Float values. Each instruction specifies its rounding mode so these instructions are not affected by the Rounding Mode in the Control Register.

An integral source value produces the same value for the destination (ignoring any saturation). For magnitudes >= 8,388,608 (2^{23}) all Float values are integral.

The rounding instructions are sign preserving.

Signed zeros are propagated. In IEEE mode, signed infinities are propagated. In IEEE mode, sNaN inputs are quietized, the equivalent qNaN is produced, and the Invalid Operation exception is indicated. In IEEE mode, qNaN inputs are propagated.

No other exceptions are signaled for these instructions. For example if the source and result values differ, the Inexact exception is not signaled.

The Single Precision Denorm Mode in the Control Register affects the results of the *rndd* and *rndu* instructions for denorm source values.

Floating-Point to Integer Conversion

The *mov* and *sel* instructions can be used to convert floating-point values to integers. In the tables below, Imin is the smallest representable value in a signed integer type, Imax is the largest representable value

in an integer type, and f is a finite floating-point value after rounding to an integral value using the current rounding mode.

Converting unrepresentable floating-point values, including infinities, NaNs, and values that convert to integers outside of the destination type's range, signal Invalid Operation exceptions. When the destination integer type is unsigned, normalized nonzero negative inputs signal Invalid Operation exceptions and negative denorm inputs signal Inexact exceptions.

Data written in accumulator using implicit or explicit destination will not be IEEE compliant.

Floating-Point to Integer Conversion Results and Exceptions for Signed Integer Types

FP Value	Integer Result	FP Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
Imin <= f <= Imax	f	Inexact if rounding changed f
f < Imin	Imin	Invalid Operation
-inf	Imin	Invalid Operation

Floating-Point to Integer Conversion Results and Exceptions for Unsigned Integer Types

FP Value	Integer Result	FP Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
0 <= f <= Imax	f	Inexact if rounding changed f
f = -0	0	Inexact
-1 < f < -0	0	Inexact
-1 < f < -0	Real Indefinite	Invalid Operation
-fmax <= f <= -1	Real Indefinite	Invalid Operation
-inf	Real Indefinite	Invalid Operation

Note: Real Indefinite is encoded as Integer value 0.

Integer to Floating-Point Conversion

Integer to floating-point conversion follows these rules in IEEE mode:

- The result is never ± inf, never NaN, and never -0.

- If the integer source value is not exactly representable in the destination floating-point format, use the current rounding mode to choose an adjacent floating-point value and signal the Inexact Exception.
- If the integer source value is too large to represent in the destination floating-point format (only possible when converting to Half Float from D, UD, or UW) then signal the Overflow Exception. Based on the sign and the current rounding mode, the result is ± fmax or ± inf, as described in the Overflow Exception section.

Floating-Point Min/Max Operations

In the following Min/Max operations, sNaN inputs are preferred to non-NaN inputs and non-NaN inputs are preferred to qNaN inputs.

Min(x, qNaN) = Min(qNaN, x) = x with no exceptions signaled.

Min(x, sNaN) = Min(sNaN, x) = qNaN (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

Min(x, sNaN) = Min(sNaN, x) = x

Max(x, qNaN) = Max(qNaN, x) = x with no exceptions signaled.

Max(x, sNaN) = Max(sNaN, x) = qNaN (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

Max(x, sNaN) = Max(sNaN, x) = x

Special cases(when both sources are NaN inputs)

Min(qNaN, qNan) = qNaN (of the first source) and no exception raised

Min(qNaN, sNaN) = Min(sNaN, qNaN) = qNaN (quiet-ized sNaN and signal the Invalid Operation exception)

Min(sNaN, sNaN) = qNaN(of the first source and signal Invalid Operation Exception)

Max(qNaN, qNan) = qNaN (of the first source) and no exception raised

Max(qNaN, sNaN) = Max(sNaN, qNaN) = qNaN (quiet-ized sNaN and signal the Invalid Operation exception)

Max(sNaN, sNaN) = qNaN(of the first source and signal Invalid Operation Exception)

IEEE Floating-Point Exceptions

The EU detects the five floating-point exceptions defined by IEEE Standard 754:

Invalid Operation

Division by Zero

Overflow

Underflow

Inexact

Signaling Floating-Point Exceptions

When enabled in the Control Register, floating-point exceptions are detected and set *sticky* flag bits in the State Register. There is no mechanism for automatic transfer to a handler, so floating-point exceptions are not handled like other exceptions described in the Exceptions chapter.

Setting flags to indicating Floating-Point exceptions is the default exception handling approach specified by IEEE Standard 754. The flag bits are sticky because in normal operation the EU only sets these bits as exceptions occur, and does not clear these bits, so a set value sticks until cleared by software. The Control Register and State Register are cleared at reset and initialized at thread load. Both are read/write registers. These fields are used:

- **IEEE Exception Trap Enable** (Control Register cr0.0:ud bit 9). This bit enables trapping IEEE exception flags. This control bit may be updated by software. It is initially zero on thread load. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exception field of sr0.1, in the State Register. **Note:** Software must set this flag at thread start to use the IEEE Exception flags.
- **IEEE Exception**. (State Register sr0.1:ud bits 7:0). The IEEE exception bits are sticky bits set by the opcodes when floating-point exceptions are triggered. These bits are defined per thread and all channels update one set of sticky bits. These bits are cleared on thread load and may be cleared by software. Exception updates to these bits may be disabled by clearing the IEEE Exception Trap Enable bit in the Control Register. The following table specifies the IEEE exception bits:

Bits	Definition
7:5	Reserved
4	Inexact Exception
3	Overflow Exception
2	Underflow Exception
1	Division by Zero Exception
0	Invalid Operation Exception

The IEEE exception flags are 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:

+inf - (+inf) or (-inf) - (-inf)

± 0 / ± 0

± inf / ± inf

± 0 × ± inf or ± inf × ± 0

Remainder(± *x*, ± 0)

Remainder(± inf, ± *x*)

Sqrt(-*x*)

Note that Sqrt(-0) is -0 per IEEE Standard 754 and does not cause any exception.

These instructions can signal specific Invalid Operation exceptions (and also on sNaN inputs except for Float inputs in ALT mode), producing a qNaN result:

add

dp2, *dp3*, *dp4*, and *dph* (the Dot Product instructions)

line

lrp

mac

mad

madm

math with the FDIV, SQRT, or RSQRTM function codes

mul

pln

These other instructions signal Invalid Operation exceptions on sNaN inputs (except for Float inputs in ALT mode), producing a qNaN result:

cmp, *cmpn*

frc

math with all other function codes for floating-point operations

mov except for raw moves

rndd, *rnde*, *rndu*, and *rndz* (the Round instructions)

sel

smov except for raw moves

Division by Zero Exception

The operation ± *x* / ± 0, where *x* is a positive, finite, nonzero, and normalized number, signals the Division by Zero Exception and produces a correctly signed ± inf result. Note that in accordance with the standard, ± 0 / ± 0 signals Invalid Operation, does not signal Division by Zero, and produces a qNaN result. This behavior can occur for the *math* instruction with the FDIV function code.

The operation $LOG_2(\pm 0)$ signals the Division by Zero Exception and produces -inf as the result. This behavior can occur for the *math* instruction with the LOG function code.

Overflow Exception

A floating-point *overflow* occurs when an operation with a finite result produces an internal result with magnitude > fmax, the maximum representable finite value in the destination format. The internal result is rounded to the destination precision with the current rounding mode but has an unbounded exponent. An overflow produces ± inf or ± fmax as the result, depending on the sign and the rounding mode.

The following algorithm describes floating-point overflow processing:

1. Compute the result with infinite precision and unbounded range.
2. Normalize the result using an unbounded exponent.
3. Round the result to the destination precision using an unbounded exponent and the current rounding mode.
4. If (abs(rounded unbounded result) > fmax(destination format)) {

   Set the Overflow Exception flag to 1.

   Output = ± inf or ± fmax depending on the sign and the rounding mode:

   + and (RNE or RU): + inf

   + and (RD or RZ): + fmax

   - and (RU or RZ): - fmax

   - and (RNE or RD): - inf

   }

   }

Note: The first three steps of the overflow and underflow algorithms are identical.

Underflow Exception

An *underflow* occurs when an operation produces a tiny but nonzero inexact result *x*, with abs(*x*) < fmin, where fmin = 2^{Emin}. See the Floating-Point Type Parameters table for the fmin and Emin values for different floating-point types.

IEEE Standard 754 allows underflow to be determined before or after rounding. The Execution Unit determines underflow after rounding, which is consistent with the behavior of the x87 and SSE instructions in the CPU.

When denorms are flushed to zero, no underflow exceptions are signaled. Flush to zero is not defined by IEEE Standard 754, but is implementation specific and required by some APIs (including DirectX), thus the EU ISA for BDW supports either using or flushing Float or Double Float denorms based on the respective Denorm Mode bits.

When denorms are enabled, if an operation's internal result rounded to the destination precision, but using an unbounded exponent range, has a magnitude that is less than fmin but nonzero AND the denorm result in the destination format is inexact, then signal the Underflow Exception.

The rounded result can be ± 0, ± fmin, or (when denorms are enabled) ± denorm.

IEEE Standard 754 requires the non-intuitive behavior that an exact denorm result does not set the Underflow Exception flag.

The following algorithm describes floating-point underflow processing:

1. Compute the result with infinite precision and unbounded range.
2. Normalize the result using an unbounded exponent.
3. Round the result to the destination precision using an unbounded exponent and the current rounding mode.
4. If (0 < abs(rounded unbounded result) < fmin(destination format) {

   if (flush denorms to zero) {

   Output = 0; // No underflow exception.

   }

   Else {

   Renormalize to the bounded exponent with the original infinite precision value...

   ...and round that value to the destination precision using the current rounding mode.

   If (the just computed value differs from the value computed in step (3) in exponent or mantissa) {

   Set the Underflow Exception flag to 1.

   } // Note: Underflow is not set if the tiny result is the same as when computed with an unbounded exponent.

   Output = rounded result using the destination precision and destination exponent range;

}

}

Else { // Not a tiny number.

Output = rounded number;

}

Inexact Exception

An Inexact Exception occurs when the internal unrounded result, with infinite precision and unbounded exponent range, differs from the generated result after format conversion, normalizing or denormalizing, and rounding. An Inexact Exception occurs irrespective of any saturation to exact zero or exact +1.0. The Inexact Exception is normal and may occur more often than not. For example, the calculation 1.0 / 3.0 is inexact in any binary floating-point format. These rules determine whether a result is inexact:

- Infinities and NaNs are never inexact.
- Flushing a denorm internal result to zero (always for Half Float and if the appropriate Denorm Mode is 0 for Float and Double Float) is always inexact.
- If any rounding occurs using the current Rounding Mode, so the rounded result differs from the internal unrounded result, the result is inexact. However explicit round to integral using any of the rounding instructions (*rndd*, *rnde*, *rndu*, and *rndz*) is never inexact.

Floating-Point Compare Operations

Four mutually exclusive relations are possible between two floating-point values, src0 and src1:

Less than. src0 < src1 and neither source is NaN.

Equal. src0 = src1 and neither source is NaN.

Greater than. src0 > src1 and neither source is NaN.

Unordered. Any source is NaN.

Any NaN compares unordered to any value, including itself.

Infinities of the same sign compare as equal.

Zeros compare as equal regardless of sign: -0 = +0.

Floating-Point Compare Relations

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	E	L	L	L	L	L	L	L	U
-fin	G	*	L	L	L	L	L	L	U
-denorm	G	G	*^	L^	L^	L^	L	L	U
-0	G	G	G^	E	E	L^	L	L	U
+0	G	G	G^	E	E	L^	L	L	U
+denorm	G	G	G^	G^	G^	*^	L	L	U
+fin	G	G	G	G	G	G	*	L	U
+inf	G	G	G	G	G	G	G	E	U
NaN	U	U	U	U	U	U	U	U	U

Notes	
*	Relation can be L, E, or G.
^	When denorms are flushed to zero then all denorms and zeros compare as E.

The next six tables show the results of six specific comparisons, corresponding to the **.g**, **.l**, **.e**, **.ne**, **.ge**, and **.le** conditional modifiers used with the *cmp* instruction and a floating-point source type. Any NaN source produces a false comparison result for these modifiers other than **.ne** and produces a true comparison result for the **.ne** modifier.

Results of Greater Than Comparison — *cmp*.g

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	F	F	F^	F	F	F
+0	T	T	T^	F	F	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	F	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.g**

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	F	F	F^	F	F	F
+0	T	T	T^	F	F	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	F	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
	comparison result F.

Results of Less Than Comparison — *cmp*.l

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
-denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	F	F	T^	T	T	F
+0	F	F	F^	F	F	T^	T	T	F
+denorm	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	F	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.l** comparison result F.

Results of Equal Comparison — *cmp*.e

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	F	*	F	F	F	F	F	F	F
-denorm	F	F	*^	F^	F^	F^	F	F	F
-0	F	F	F^	T	T	F^	F	F	F
+0	F	F	F^	T	T	F^	F	F	F
+denorm	F	F	F^	F^	F^	*^	F	F	F
+fin	F	F	F	F	F	F	*	F	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.e** comparison result T.

Results of Not Equal Comparison — *cmp*.ne

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	T
-fin	T	*	T	T	T	T	T	T	T
-denorm	T	T	*^	T^	T^	T^	T	T	T
-0	T	T	T^	F	F	T^	T	T	T
+0	T	T	T^	F	F	T^	T	T	T
+denorm	T	T	T^	T^	T^	*^	T	T	T
+fin	T	T	T	T	T	T	*	T	T
+inf	T	T	T	T	T	T	T	F	T
NaN	T	T	T	T	T	T	T	T	T

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.ne** comparison result F.

Results of Greater Than or Equal Comparison — *cmp*.ge

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	T	T	F^	F	F	F
+0	T	T	T^	T	T	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes

*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.ge** comparison result T.

Results of Less Than or Equal Comparison — *cmp*.le

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
-denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	T	T	T^	T	T	F
+0	F	F	F^	T	T	T^	T	T	F
+denorm	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes

*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.le** comparison result T.

Type Conversion

This topic is currently under development.

Float to Integer

Converting from float to integer is based on rounding toward zero(RTZ is for DX, IEEE expects all four rounding modes). If the floating point value is +0, -0, +Denorm, -Denorm, +NaN –r -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UD, D, UW, W, UB and B).

Input Format	Output Format					
F	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

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 GEN Architecture defines a basic exception handling mechanism for several exception cases. This mechanism supports both normal operations such as extensions of the mask-stack depth, as well as detecting some illegal conditions.

Exception Types

Type	Trigger / Source	Sync/Async Recognition
Software Exception	Thread code	Synchronous
Breakpoint	• A bit in the instruction word • Breakpoint IP match • Breakpoint Opcode match	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous
Context Save/Restore	Preemption Interrupt	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver, and delivered to the EU as part of a new thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's cr0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception type recognition across hardware threads. The exception 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.

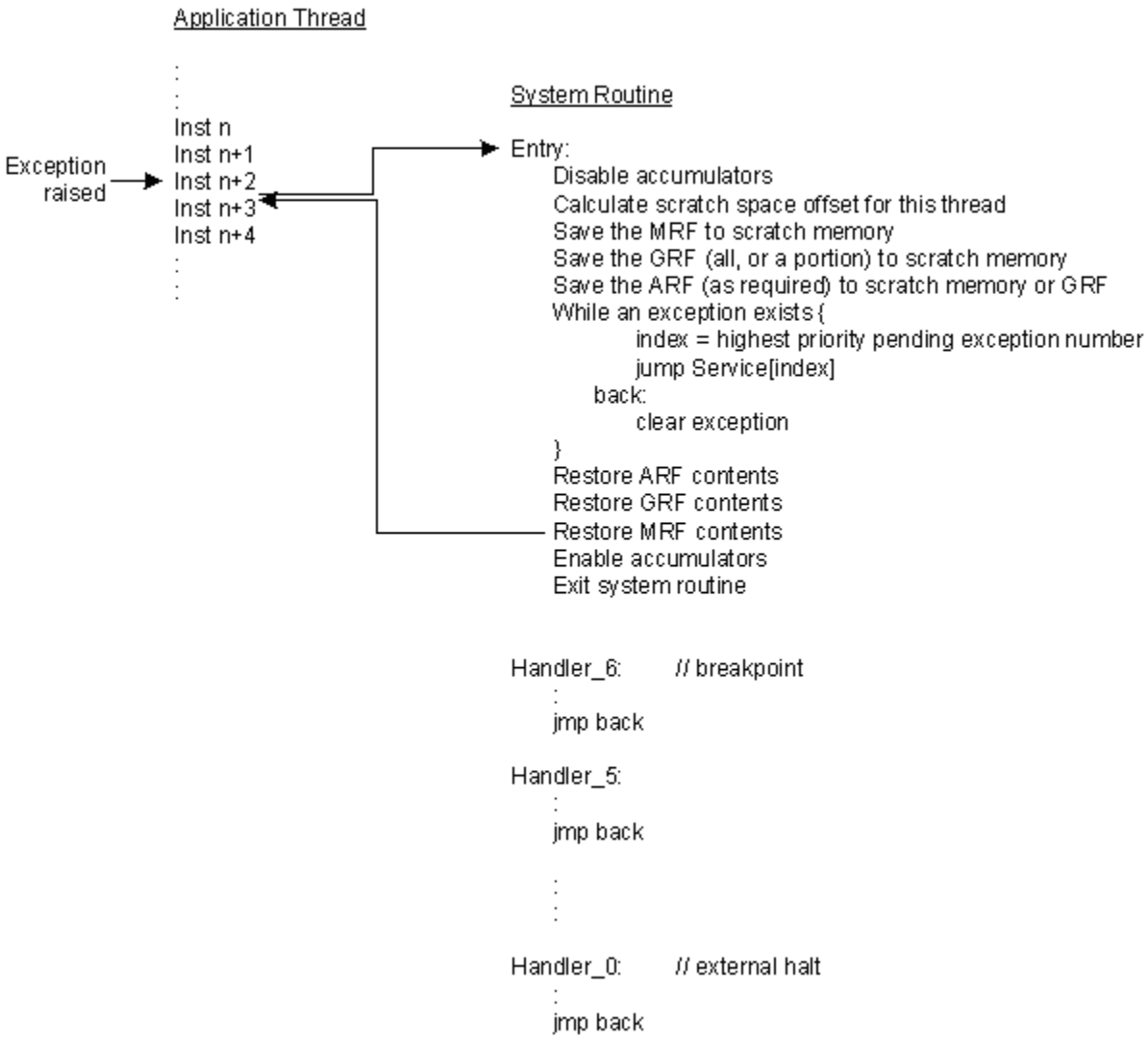
Exception-Related Architecture Registers

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 Master 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 Master Exception State and Control bit with at least one SIMD-16 *nop* instruction to allow control to transition. Throughout the System Routine, the AIP register maintains its value at the time the exception was raised unless directly modified by the System Routine. (See the AIP register definition for specifics on the IP value saved to AIP).

System IP (SIP)

The System IP (SIP) is the 16 byte-aligned offset of the first instruction of the System Routine, relative to the General State Base Address. SIP is assigned by the STATE_SIP command to the command streamer which updates SIP in the EU.

The SIP is widened to 48 bits. However the EU still only uses the low 32 bits (bits 31:4 with bits 3:0 as zero).

When the System Routine is invoked, the application thread's current IP is first saved into the AIP field of cr0.2. The SIP address is then loaded into the thread's IP register and execution continues within the System Routine. Thus each invocation of the System Routine has a common entry point. Returning from the System Routine loads IP from AIP, continuing thread execution.

System Routine Register Space

The System Routine uses the same GRF space as the thread that invokes it.

As such all of the calling thread's registers and their contents are visible to the System Routine. Further, the System Routine must only use r0..r15 of the GRF, as a minimal thread may have requested and been allocated this few. If the System Routine requires more registers than this, the driver should establish a higher minimum allocation for all threads.

The System Routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

Only one 32-bit GRF location, r0.4, is reserved for System Routine use. This location is sufficient to allow the System Routine to calculate the appropriate offset of its private scratch memory in the larger system scratch memory space (as dictated by binding table entry 254). The offset is left as a driver convention, but is likely based on a combination of Thread and EU IDs (see the example system handler in the *System Scratch Memory Space*) section. Other than the reserved r0.4 register field, there is no explicit GRF register space dedicated to the System Routine, and any GRF needs must be accomplished via (a) convention between the System Routine and application code, or (b) the System Routine temporarily spilling the thread's GRF register contents to scratch memory and restoring those contents before System Routine exit.

No persistent storage is automatically allocated to the System Routine, although a driver implementation may set aside part of system scratch memory for the System Routine.

Any parameter passing to the System Routine (for use by software exceptions) is done via the GRF based on system thread/application thread convention.

System Scratch Memory Space

There is a single unified system scratch memory space per context shared by all EUs. It is anticipated that block is further partitioned into a unique scratch sub-space per thread via conventions implemented in the System Routine, with each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread can be based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architecture register sr0.0.

Per_Thread_Block_Size = System_Scratch_Block_Size / (EU_Count * Thread_Per_EU);

Offset = (sr0.0.EID * Threads_Per_EU + sr0.0.TID) * Per_Thread_Block_Size;

where in GEN:

Threads_Per_EU = 4

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.

Use of NoDDClr

The GEN instruction word defines an instruction option **NoDDClr** that overrides the native register dependency clearing mechanism of the typical instruction. When specified, NoDDClr does not clear, at register writeback time, the dependency placed on the destination register of the instruction. Use of this mechanism may provided increased performance when a kernel can guarantee no dependency issues between instructions, but may cause issues with exception handling in some circumstances as discussed here.

Typically NoDDClr is used in an instruction series to enable a sequence of writes to sub-fields of a GRF register without paying a dependency penalty on each instruction. In this case, NoDDClr and NoDDChk are used across an instruction sequence to allow the first instruction to set the destination dependency, interior instructions to write to the GRF register without dependency checks, and the last instruction to clear the dependency. (This sequence is referred to as a NoDDClr code block going forward). By only allowing the last instruction to clear the dependency, program execution is prevented from going beyond a certain point until all writes of that sequence are known to retire.

The problem arises if an exception is raised within a NoDDClr code block. In this case, there exists the potential for the System Routine to hang while attempting to save/restore a register used as a destination register by the NoDDCLr code block, as the outstanding dependency on that register will not clear until the final instruction of the NoDDCLr block is executed, sometime after the System Routine returns. Should the System Routine attempt to use that register, it hangs waiting on a dependency to be cleared by an instruction not yet issued.

Note: This is a known condition and will in some cases not allow the full GRF contents to be externally visible in System Routine scratch space during a break or halt exception.

To avoid this condition, guidelines are provided below for consideration. (Note that these are general guidelines, some of which can be alleviated through careful coding and register usage conventions and restrictions.)

Guideline
NoDDClr code blocks should only be used where absolutely necessary.
Instructions that may generate exceptions should not be placed within NoDDClr blocks. This includes most conditional branch instructions (if, do, while, ...).
If possible, use NoDDClr on registers high in the thread's register allocation (e.g. r120), thus even if a System Routine hang occurs, as much of the GRF is visible as possible. (Note that this would also require the System Routine to update the progress of the GRF dump, perhaps with each GRF block written, or to initialize the System Routine's scratch space to a known value, to be able to distinguish valid/locations from unwritten locations).

Also a driver implementation may consider a disable-NoDDclr option in which jitted code does not use the NoDDClr capability. In this case, there is no change to the code that is jitted other than removal of

the NoDDClr instruction option. The code executes as normal, but with a higher number of thread switches in what would have been a NoDDClr code block.

Exception Descriptions

This section describes conditions that can cause exceptions and transfer control to the System Routine.

Illegal Opcode

The GEN 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 master 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 master 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 master 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 master 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.

Note that GEN does not support self-modifying code, therefore the driver has an opportunity to detect such cases before the thread is placed in service.

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 SNB code for the message registers in the MRF. All message register and MRF references are specific to SNB. Other code in this example is useful for other processor generations.

The example frees enough MRF and GRF space to get the routine started, then jumps to the handler for the specific exception. Many other implementations are also valid, including single exception servicing (as opposed to looping) per invocation, and saving only the GRF or MRF space required by the exception being serviced.

```
#define ACC_DISABLE_MASK 0xFFFFFFFD
#define MASTER_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           // MRF is specific to SNB.
#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). MRF is specific to SNB:
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). MRF is specific to
SNB.
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 Use of NoDDClr mov (16) m1 r0 {NoMask}
send (8) m0 null null DPW|ML2|RL0 {NoMask}

// Save r2..r3 to free some room:
mov (16) m3 r2 {NoMask}
```

```
add (1) m0.2 r0.4 SCR_OFFSET_GRF + 64 {NoMask}
send (8) m0 null null DPW|ML4|RL0 {NoMask}

// Save r4..r7 to free some room (optional, depending on needs):
mov (16) m8 r4 {NoMask}
mov (16) m10 r6 {NoMask}
add (1) m7.2 r0.4 (SCR_OFFSET_GRF + 128) {NoMask}
send (8) m7 null null DPW|ML5|RL0 {NoMask}

// Save r8..r11 to free some room (optional, depending on needs):
mov (16) m1 r8 {NoMask}
mov (16) m3 r10 {NoMask}
add (1) m0.2 r0.4 (SCR_OFFSET_GRF + 256) {NoMask}
send (8) m0 null null DPW|ML5|RL0 {NoMask}

// Save r12..r15 to free some room (optional, depending on needs):
mov (16) m8 r12 {NoMask}
mov (16) m10 r14 {NoMask}
add (1) m7.2 r0.4 (SCR_OFFSET_GRF + 384) {NoMask}
send (8) m7 null null DPW|ML5|RL0 {NoMask}

// Save selected ARF registers (optional, depending on use):
// flags, others ...
// Save f0.0:
mov (1) r1.0:uw f0.0 {NoMask}

Next:  // Exceptions pending? If not, exit.

cmp.e (1) f0.0 cr0.4:uw 0:uw {NoMask}
(f0.0) mov (1) IP EXIT {NoMask}

// Find highest priority exception:
lzd (1) r1.1:uw cr0.4:uw {NoMask}

// Jump table to service routine:
jmpi (1) r1.1:uw{NoMask}
mov (1) IP CRService_0 {NoMask}
mov (1) IP CRService_1 {NoMask}
mov (1) IP CRService_2 {NoMask}
...
mov (1) IP CRService_15{NoMask}
mov (1) IP Next

Service_0:
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).
...
Service_15:
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).

Exit:
// Restore f0.0.
// Restore other ARF registers (as required).
// Restore r12..r15.
// Restore r8..r11.
// Restore r4..r7.
// Restore r0..r3.
// Restore m8..m15.
// Restore m0..m7.
```

```
// Clear Master Exception State bit in cr0.0:
and (1) cr0.0 cr0.0 MASTER_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

GEN 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 GEN 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 GEN 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 GEN instruction works on twice as much SIMD data as that for a non-compressed GEN instruction. A compressed instruction is converted into two native instructions by the instruction dispatcher in the EU.

GEN instructions are executed on a narrower SIMD execution pipeline. Therefore, GEN native instructions take multiple execution cycles to complete.

Instruction Operands and Register Regions

Most GEN 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.

Description
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.

Instruction Formats

This section shows the machine formats of the GEN instruction set. The instructions in the GEN architecture have a fixed length of 128 bits in the native format. A compact format, discussed separately in this volume, can represent some instructions using 64 bits. Out of the 128 bits in the native format, there are 120 bits in use, and the remaining bits are reserved for future extensions. One instruction

consists of instruction fields that control various stages of execution. These fields are roughly grouped into the 4 DWords as follows:

- Instruction Operation Doubleword (DW0) contains the Opcode and other general instruction control fields.
- Instruction Destination Doubleword (DW1) specifies the destination operand (dst) and the register file and type of source operands.
- Instruction Source 0 Doubleword (DW2) contains the first source operand (src0).
- Instruction Source 1 Doubleword (DW3) contains the second source operand (src1) and is used to hold any 32-bit immediate source (Imm32 as src0 or src1).

Instructions with one source operand of type DF, Q, or UQ can use an Imm64 64-bit immediate source operand in DWord 2 and DWord3.

Most instructions have 1 or 2 source operands and use a common instruction format. Within that format, there are variations based on AddrMode and AccessMode. There is a separate instruction format for a small number of instructions with 3 source operands. Send, math, and branching instructions have format variations described separately.

The 3-source instructions have the following restrictions:

- Only GRF registers can be sources and only GRF registers can be the destination.
- Subregister numbers have DWord granularity if AccessMode is Align16.
- AccessMode is Align16, uses Align16-style swizzling, with extra replication control. There is no other regioning support.

The next two subsections describe the instruction formats for various processor generations using tables. The following diagrams provide another view of the same information. The first diagram is for native instructions with one or two source operands.

GEN Instruction Format – 1-src and 2-src

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	AddrMode = Direct: AccessMode = Align16	AddrMode = Direct: AccessMode = Align1	AddrMode = Indirect: AccessMode = Align16	AddrMode = Indirect: AccessMode = Align1	SEND: MsgDesc Imm	SEND: MsgDesc Reg	MATH	Branch (2offsets)	Branch (1offset)	Imm Src[31:0]	Imm Src[63:0]
	1	127	127	1					EOT						
	6	126	121	6											
	4	120	117	4	Src1.VertStride										
	1	116	116	1											
	2	115	114	2		Src1.Width		Src1.Width							
	2	113	112	2	Src1.ChanSel[7:4]	Src1.HorzStride	Src1.ChanSel[7:4]	Src1.HorzStride							
	1	111	111	1	Src1.AddrMode										
	2	110	109	2	Src1.SrcMod										
	4	108	105	4			Src1.AddrSubRegNum								
	4	104	101	4	Src1.RegNum [7:0]										
	1	100	100	1	Src1.SubRegNum [4]		Src1.AddrImm[8:4]								
3	4	99	96	4	Src1.ChanSel[3:0]	Src1.SubRegNum [4:0]	Src1.ChanSel[3:0]	Src1.AddrImm[8:0]			Same	JIP[31:0]	JIP[31:0]	Imm[31:0]	
	1	95	95	1											
	4	94	91	4	Src1.SrcType										
	2	90	89	2	Src1.RegFile										
	4	88	85	4	Src0.VertStride										
	1	84	84	1											
	2	83	82	2		Src0.Width		Src0.Width							
	2	81	80	2	Src0.ChanSel[7:4]	Src0.HorzStride	Src0.ChanSel[7:4]	Src0.HorzStride							
	1	79	79	1	Src0.AddrMode										
	2	78	77	2	Src0.SrcMod										
	4	76	73	4			Src0.AddrSubRegNum								
	4	72	69	4	Src0.RegNum [7:0]		Src0.AddrImm [8:4]								
	1	68	68	1	Src0.SubRegNum [4]										
2	4	67	64	4	Src0.ChanSel[3:0]	Src0.SubRegNum [4:0]	Src0.ChanSel[3:0]	Src0.AddrImm [8:0]				UIP[31:0]			Imm[63:0]
	1	63	63	1	Dst.AddrMode										
	2	62	61	2		Dst.HorzStride		Dst.HorzStride							
	4	60	57	4			Dst.AddrSubRegNum								
	4	56	53	4	Dst.RegNum [7:0]										
	1	52	52	1	Dst.SubRegNum [4]		Dst.AddrImm [8:4]								
	4	51	48	4	Dst.ChanEn[3:0]	Dst.SubRegNum [4:0]	Dst.ChanEn[3:0]	Dst.AddrImm [8:0]							
	1	47	47	1	Reserved										
	4	46	43	4	Src0.SrcType										
	2	42	41	2	Src0.RegFile										
	4	40	37	4	Dst.Type										
	2	36	35	2	Dst.RegFile										
	1	34	34	1	WECtrl										
	2	33	32	2	FlagRegNum/FlagSubRegNum				Same		Same	Same	Same	Same	Same
	1	31	31	1	Saturate										
	1	30	30	1	DebugCtrl										
	1	29	29	1	CmptCtrl										
	1	28	28	1	AccWrCtrl / BranchCtrl				Same		Same	Same	Same		
	4	27	24	4	CondModifier				SFID[3:0]		FC[3:0]	MBZ	MBZ		
	3	23	21	3	ExecSize										
	1	20	20	1	PredInv										
	4	19	16	4	PredCtrl										
	2	15	14	2	ThreadCtrl										
	2	13	12	2	QtrCtrl										
	1	11	11	1	NibCtrl										
	2	10	9	2	DepCtrl										
	1	8	8	1	AccessMode				Same		Same	Same	Same	Same	Same
	1	7	7	1	(reserved for Opcode)										
0	7	6	0	7	Opcode				Same		Same	Same	Same	Same	Same

The next diagram is for native instructions with three source operands.

GEN Instruction Format – 3-src

DW#	Instr Bits	High Bit	Low Bit	Instr Bits	Description
	1	127	127	1	*reserved*
	1	126	126	1	*reserved*
	8	125	118	8	Src2 Regnum
	3	117	115	3	Src2 Subregnum
	8	114	107	8	Src2 Swizzle
	1	106	106	1	Src2 RepCtrl
	1	105	105	0	*reserved*
	8	104	97	8	Src1 Regnum
	3	96	94	3	Src1 Subregnum
	8	93	86	8	Src1 Swizzle
	1	85	85	1	Src1 RepCtrl
	1	84	84	0	*reserved*
	8	83	76	8	Src0 Regnum
	3	75	73	3	Src0 Subregnum
	8	72	65	8	Src0 Swizzle
2,3	1	64	64	1	Src0 RepCtrl
	8	63	56	8	Dst Regnum
	3	55	53	3	Dst Subregnum
	4	52	49	4	Dst chan enable
	3	48	46	3	Dst Type
	3	45	43	3	Src Type
	2	42	41	2	Src2 Modifier
	2	40	39	2	Src1 Modifier
	2	38	37	2	Src0 Modifier
	1	36	36	1	*reserved*
	1	35	35	1	*reserved*
	1	34	34	1	*WECtrl*
1	2	33	32	2	*FlagRegNum/FlagSubRegNum*
	1	31	31	1	*Saturate*
	1	30	30	1	*DebugCtrl*
	1	29	29	1	*CmptCtrl*
	1	28	28	1	*AccWrCtrl*
	4	27	24	4	*CondModifier*
	3	23	21	3	*ExecSize*
	1	20	20	1	*PredInv*
	4	19	16	4	*PredCtrl*
	2	15	14	2	*ThreadCtrl*
	2	13	12	2	*QtrCtrl*
	1	11	11	1	*NibCtrl*
	2	10	9	2	*DepCtrl*
	1	8	8	1	*AccessMode*
	1	7	7	0	*(reserved for Opcode)*
0	7	6	0	7	*Opcode*

GEN Instruction Format – 3-src

DW#	Instr Bits	High Bit	Low Bit	Instr Bits	Description
	1	127	127	1	*reserved*
	1	126	126	1	Src2 Subregnum
	8	125	118	8	Src2 Regnum
	3	117	115	3	Src2 Subregnum
	8	114	107	8	Src2 Swizzle
	1	106	106	1	Src2 RepCtrl
	1	105	105	1	Src1 Subregnum
	8	104	97	8	Src1 Regnum
	3	96	94	3	Src1 Subregnum
	8	93	86	8	Src1 Swizzle
	1	85	85	1	Src1 RepCtrl
	1	84	84	1	Src0 Subregnum
	8	83	76	8	Src0 Regnum
	3	75	73	3	Src0 Subregnum
	8	72	65	8	Src0 Swizzle
2,3	1	64	64	1	Src0 RepCtrl
	8	63	56	8	Dst Regnum
	3	55	53	3	Dst Subregnum
	4	52	49	4	Dst chan enable
	3	48	46	3	Dst Type
	3	45	43	3	Src Type
	2	42	41	2	Src2 Modifier
	2	40	39	2	Src1 Modifier
	2	38	37	2	Src0 Modifier
	1	36	36	1	Src1 Type
	1	35	35	1	Src2 Type
	1	34	34	1	*WECtrl*
1	2	33	32	2	*FlagRegNum/FlagSubRegNum*
	1	31	31	1	*Saturate*
	1	30	30	1	*DebugCtrl*
	1	29	29	1	*CmptCtrl*
	1	28	28	1	*AccWrCtrl*
	4	27	24	4	*CondModifier*
	3	23	21	3	*ExecSize*
	1	20	20	1	*PredInv*
	4	19	16	4	*PredCtrl*
	2	15	14	2	*ThreadCtrl*
	2	13	12	2	*QtrCtrl*
	1	11	11	1	*NibCtrl*
	2	10	9	2	*DepCtrl*
	1	8	8	1	*AccessMode*
	1	7	7	0	*(reserved for Opcode)*
0	7	6	0	7	*Opcode*

Instruction Fields

This section contains two large tables that together describe all instruction fields. The first table describes common instruction fields that are not specific to a source or destination operand. The second table describes fields used to describe source or destination operands, including fields that describe register regioning and immediate operand fields.

Notable changes for BDW+ instructions include adding three new data types (Q, UQ, HF), widening all type fields to four bits each to accomodate the new type encodings, providing a new source modifier interpretation for logical operations, widening the AddrSubRegNum fields to four bits to accomodate 16 (rather than 8) address subregisters, and supporting 64-bit immediate (Imm64) source values for instructions with single source operands.

In the assembler syntax, some fields appear in the positions used for destination or source operands but are not normal operands. Such fields appear in the Common Instruction Fields table, notably the JIP and UIP instruction offsets used in some flow control instructions.

Common Instruction Fields (Alphabetically by Short Name)

Field	Description
AccessMode	**Access Mode.** This field controls operand access for the instruction. It applies to all source and destination operands. When it is clear (**Align1**), the instruction uses byte-aligned addressing for source and destination operands. Source swizzle control and destination mask control are not supported. Elements must still be aligned on element-size boundaries. For example, a DWord operand must be aligned on a 4-byte boundary. When it is set (**Align16**), the instruction uses 16-byte-aligned addressing for source and destination operands. Source swizzle control and destination mask control are supported in this mode. A register region must start on 16-byte aligned address, but individual elements do not. For example, a packed array of 16 bytes can be an operand, where only the first byte, at offset 0, is at a 16-byte aligned address. 0 = **Align1** 1 = **Align16**
AccWrCtrl/ BranchCtrl	**Accumulator Write Control.** Enable or disable implicitly writing results to the accumulator as well as to the destination. 0 = Do not write results to the accumulator; write results only to the destination. 1 = **AccWrEn.** Write results to the accumulator as well as to the destination. This bit should not be set if the accumulator is the explicit destination operand. Some instructions do not allow this option, including *mov* and *send*. **Branch Control.** Used by the *goto* instruction to control branching. See the **goto** instruction description for more information about BranchCtrl.
CmptCtrl	**Compaction Control.** Indicates whether the instruction is compacted to the 64-bit compact instruction format. When this bit is set, the 64-bit compact instruction format is used. The EU decodes the compact format using lookup tables internal to the hardware, but documented for use by software tools. Only some instruction variations can be compacted, the variations supported by those lookup tables and the compact format. See EU Compact Instruction Format for more information. 0 = No Compaction. No compaction. 128-bit native instruction supporting all instruction options. 1 = **CmptCtrl**. Compaction is enabled. 64-bit compact instruction supporting only some instruction variations.

Field	Description
CondModifier	**Condition Modifier.** This field sets the flag register based on internal conditional signals output from the execution pipe such as sign, zero, overflow, NaNs, etc. If this field is 0000b, no flag registers are updated. Flag registers are not updated for instructions with embedded compares. This field applies to all instructions except send, sendc, and math. 0000b = Do not modify the flag register (normal) 0001b = Zero or Equal (**.z** or **.e**) 0010b = Not Zero or Not Equal (**.nz** or **.ne**) 0011b = Greater than (**.g**) 0100b = Greater than or equal (**.ge**) 0101b = Less than (**.l**) 0110b = Less than or equal (**.le**) 0111b = Reserved 1000b = Overflow (**.o**) 1001b = Unordered with Computed NaN (**.u**) 1010b-1111b = Reserved
DepCtrl	**Destination Dependency Control.** This field selectively disables destination dependency check and clear for this instruction. When it is set to 00b, normal destination dependency control is performed for the instruction; hardware checks for destination hazards to ensure data integrity. Specifically, a destination register dependency check is conducted before the instruction is made ready for execution. After the instruction is executed, the destination register scoreboard is cleared when the destination operands retire. When **NoDDClr** is set, the destination register scoreboard is *not* cleared when destination operands retire. When **NoDDChk** is set, hardware does not check for destination register dependencies before the instruction is made ready for execution. **NoDDClr** and **NoDDChk** are not mutually exclusive; both can be set. When this field is not 00b, hardware does not protect against destination hazards for the instruction. These settings are typically used to assemble data in a fine grained fashion (for example, a matrix-vector compute with dot-product instructions), where data integrity is guaranteed by software based on the intended usage of instruction sequences. 00b = Destination dependency checked and cleared (normal). 01b = **NoDDClr**. Destination dependency checked but not cleared. 10b = **NoDDChk**. Destination dependency not checked but cleared. 11b = **NoDDClr, NoDDChk**. Destination dependency not checked and not cleared. See the Destination Hazard and the Use of NoDDClr sections for more information.

Field	Description
EOT	**End of Thread.** For a *send* or *sendc* instruction, this bit controls thread termination. It is not used for other instructions. For a *send* or *sendc*instruction, if this field is set, the EU terminates the thread and also sets the EOT bit in the message sideband. 0 = The thread is not terminated. 1 = End of thread (EOT).
ExDesc	**Extended message descriptor.** A 32 bit immediate extended message descriptor for *send* and *sendc* instructions. This field is not used for other instructions.
ExecSize	**Execution Size.** Specifies the number of parallel execution channels and data elements for the instruction, a power of 2 from 1 to 32. The size cannot exceed the maximum number of channels allowed for the largest data type specified for a source or destination operand. *ExecSize* x largest data type size in bytes <= 64. 000b = 1 Channels (scalar operation). 001b = 2 Channels. Any data type. 010b = 4 Channels. Any data type. 011b = 8 Channels. Any data type. 100b = 16 Channels. 4-byte or smaller data types. Excludes DF, Q, and UQ types. 101b = 32 Channels. 2-byte or 1-byte data types. Excludes D, DF, F, Q, UD, and UQ types. 110b-111b = Reserved. An operand's *Width* must be <= *ExecSize*.
FC	**Function Control.** Specifies the extended math function carried out by the *math*instruction. Not used for any other instruction. This field is in the same position as the CondModifier instruction field, so the *math* instruction does not support conditional modifiers. See the *math* Extended Math Function instruction description for more information about this field.
FlagRegNum/ **FlagSubRegNum**	**Flag Register Number/Flag Sub-Register Number.** Selects the 32-bit flag register (f0 or f1) and the 16-bit flag subregister (.0 or .1). The specified flag subregister is the source for any predication and the destination for new flag values produced by any enabled conditional modifier. A flag subregister can be both a predication source and a conditional modifier destination in the same instruction. The number of flag bits used or updated depends on the execution size. 00b f0.0 01b f0.1 10b f1.0 11b f1.1
JIP	**Jump Instruction Pointer.** A Doubleword Signed Integer offset relative to the current IP (which references the current instruction) in units of bytes. Typically an immediate value in the instruction. For the *brc* (Branch Converging) instruction, both JIP and UIP can be contained in a register.

Field	Description
MaskCtrl	**Mask Control** (formerly WECtrl/Write Enable Control). This flag disables the normal write enables; it should normally be 0. 0 = Use the normal write enables in Dst.ChanEn (normal setting). 1 = NoMask. Write all channels except those disabled by predication or by other masks besides the write enables. MaskCtrl = NoMask also skips the check for PcIP[n] == ExIP before enabling a channel, as described in the Evaluate Write Enable section.
NibCtrl	**Nibble Control.** This field is used in some instructions along with QtrCtrl. See the description of QtrCtrl below. NibCtrl is only used for SIMD4 instructions with a 64bit datatype as source or destination. 0 = Use an odd 1/8th for DMask/VMask and ARF (first, third, fifth, or seventh depending on QtrCtrl). 1 = Use an even 1/8th for DMask/VMask and ARF (second, fourth, sixth, or eighth depending on QtrCtrl). Note that if eighths are given zero-based indices from 0 to 7, then NibCtrl = 0 indicates even indices and NibCtrl = 1 indicates odd indices.
NoSrcDepSet	**No Source Dependency Set.** In *send*, *sendc*, *sends* and *sendsc* instruction, this bit controls the setting of GRF source dependency. The source dependencies of both sources are considered together. When it is set to 0, normal write after read (WAR) dependency control is performed for the instruction; hardware checks for write after read hazard and ensure data integrity. Specifically source register dependency is set by the current instruction and a destination register dependency check is expected to be conducted before the next instruction is made ready for execution. When it is set to 1, hardware does not protect against write after read data hazard for the instruction. This setting is used in send instruction where the same source data is reused without modification by multiple send instruction. Software must send to the same shared function which must not be in fault and stream page fault mode and guarantee one of the following for all the source registers used in a send instruction to ensure data integrity. • The last send instruction before the instruction which modifies its source(s) does not use this setting. • The instruction which modifies the source(s) is either dependent on the destination register of the last send instruction with the source(s) or has instruction before it which does so. • The instruction which modifies the source(s) is either dependent on the source register of the last send instruction with the source(s) which does not use this setting or has instruction before it which does so. 0 = source dependency is set (normal). 1 = NoSrcDepSet. Source dependency is not set.

Field	Description
OpCode	**Operation Code.** Contains the instruction operation code. Each opcode used is given a unique mnemonic. For example, the opcode 0x01 has the mnemonic *mov* indicating a Move instruction. See the Opcode Encoding section for details of opcode encoding.
PredCtrl	**Predicate Control.** This field, together with PredInv, controls generating the predication mask (PMask) for the instruction. It allows per-channel conditional execution the instruction based on the content of the selected flag register. The encoding depends on the access mode. See the Predication section for more information about predication. In **Align16** access mode, there are eight encodings (including no predication). All encodings are based on group-of-4 predicate bits, including channel sequential, replication swizzles and horizontal any or all operations. The same configuration is repeated for each group-of-4 execution channels. In **Align1** access mode, there are twelve encodings (including no predication). The encodings apply to all execution channels with explicit channel grouping from a single channel up to a group of 16 channels. *Predicate Control in* **Align16** *access mode:* 0000b = No predication (normal). 0001b = Predication with sequential flag channel mapping. 0010b = Predication with replication swizzle **.x**. 0011b = Predication with replication swizzle **.y**. 0100b = Predication with replication swizzle **.z**. 0101b = Predication with replication swizzle **.w**. 0110b = Predication with **.any4h.** 0111b = Predication with **.all4h.** 1000b-1111b = Reserved. *Predicate Control in* **Align1** *access mode:* 0000b = No predication (normal). 0001b = Predication with sequential flag channel mapping. 0010b = Predication with **.anyv** (f0.0 OR f1.0 on the same channel). 0011b = Predication with **.allv** (f0.0 AND f1.0 on the same channel). 0100b = Predication with **.any2h** (any in group of 2 channels). 0101b = Predication with **.all2h** (all in group of 2 channels). 0110b = Predication with **.any4h** (any in group of 4 channels). 0111b = Predication with **.all4h** (all in group of 4 channels). 1000b = Predication with **.any8h** (any in group of 8 channels).

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>1001b = Predication with .all8h (all in group of 8 channels).
1010b = Predication with .any16h (any in group of 16 channels).
1011b = Predication with .all16h (all in group of 16 channels).
1100b = Predication with .any32h (any in group of 32 channels).
1101b = Predication with .all32h (all in group of 32 channels).
1110b-1111b = Reserved.</td></tr>
<tr><td>PredInv</td><td>Predicate Inverse. Together with PredCtrl, controls generation of the predication mask (PMask) for the instruction. When it is set and PredCtrl is not 0000b, predication uses the inverse of the predication bits produced by PredCtrl. In other words, the effect of PredInv happens after the effects of PredCtrl.
This field is ignored if PredCtrl is 0000b; there is no predication.
0 = +. Positive polarity of predication. Use the predication mask produced by PredCtrl.
1 = -. Negative polarity of predication. If PredCtrl is nonzero, invert the predication mask.
PMask is the final predication mask produced by the effects of both fields.</td></tr>
<tr><td>QtrCtrl</td><td>Quarter Control. This field provides explicit control for ARF selection.
This field combined with ExecSize determines which channels are used for the ARF registers.
Along with NibCtrl, 1/8 DMask/VMask and ARF can be selected.
<table>
<tr><th>QtrCtrl</th><th>NibCtrl</th><th>ExecSize</th><th>Syntax</th><th>Description</th></tr>
<tr><td>00b</td><td>0</td><td>8</td><td>1Q</td><td>Use first quarter for DMask/VMask.
Use first half for everything else.</td></tr>
<tr><td>01b</td><td>0</td><td>8</td><td>2Q</td><td>Use second quarter for DMask/VMask.
Use second half for everything else.</td></tr>
<tr><td>10b</td><td>0</td><td>8</td><td>3Q</td><td>Use third quarter for DMask/VMask.
Use first half for everything else.</td></tr>
<tr><td>11b</td><td>0</td><td>8</td><td>4Q</td><td>Use fourth quarter for DMask/VMask.
Use second half for everything else.</td></tr>
<tr><td>00b</td><td>0</td><td>16</td><td>1H</td><td>Use first half for DMask/VMask.
Use all channels for everything else.</td></tr>
<tr><td>10b</td><td>0</td><td>16</td><td>2H</td><td>Use second half for DMask/VMask.
Use all channels for everything else.</td></tr>
<tr><td>00b</td><td>0</td><td>4</td><td>1N</td><td>Use first 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>00b</td><td>1</td><td>4</td><td>2N</td><td>Use second 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>01b</td><td>0</td><td>4</td><td>3N</td><td>Use third 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>01b</td><td>1</td><td>4</td><td>4N</td><td>Use fourth 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>10b</td><td>0</td><td>4</td><td>5N</td><td>Use fifth 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>10b</td><td>1</td><td>4</td><td>6N</td><td>Use sixth 1/8th for DMask/VMask and ARF.</td></tr>
<tr><td>11b</td><td>0</td><td>4</td><td>7N</td><td>Use seventh 1/8th for DMask/VMask and ARF.</td></tr>
</table></td></tr>
</table>

Field	Description
	<table><tr><td>11b</td><td>1</td><td>4</td><td>8N</td><td>Use eighth 1/8th for DMask/VMask and ARF.</td></tr></table> **2H** is only allowed for SIMD16 instruction in Single Program Flow mode (SPF = 1). NibCtrl is only allowed for SIMD4 instructions with a 64bit datatype as source or destination type.
Reg32	In a *send*, *sendc*, *sends* or *sendsc* instruction refers to the option of providing the message descriptor field DWords correspondingly as the first dwords of the Address Register rather than as an immediate operand.
Saturate	**Saturate.** Enables or disables destination saturation. When it is set, output values to the destination register are saturated. The saturation operation depends on the destination data type. Saturation is the operation that converts any value outside the saturation target range for the data type to the closest value in the target range. For a floating-point destination type, the saturation target range is [0.0, 1.0]. For a floating-point NaN, there is no "closest value"; any NaN saturates to 0.0. Note that enabling Saturate overrides all of the NaN propagation behaviors described for various numeric instructions. Any floating-point number greater than 1.0, including +INF, saturates to 1.0. Any negative floating-point number, including -INF, saturates to 0.0. Any floating-point number in the range 0.0 to 1.0 is not changed by saturation. For an integer destination type, the maximum range for that type is the saturation target range. For example, the saturation range for B (Signed Byte Integer) is [-128, 127]. When Saturate is clear, destination values are not saturated. For example, a wrapped result (modulo) is output to the destination for an overflowed integer value. See the Numeric Data Types section for information about data types and their ranges. 0 = No destination modification (normal). 1 = **sat**. Saturate the output.
SFID	**Shared Function ID.** Specifies a shared function that is the target of a *send*or *sendc* instruction. This field is not used for any other instructions. This field is in the same position as the CondModifier instruction field, so the *send* and *sendc*instructions do not support conditional modifiers.
ThreadCtrl	**Thread Control.** This field provides explicit control for thread switching. If this field is set to 00b, it is up to the Execution Unit to manage thread switching. This is the normal operating mode. In this mode, for example, if the current instruction cannot proceed due to operand dependencies, the EU switches to the next available thread to fill the compute pipe. For another example, if the current instruction is ready to run but another thread with higher priority also has an instruction ready, the EU switches to that thread. If this field is **Switch**, a forced thread switch occurs after the current instruction executes and before the next instruction. In addition, a long delay (longer than the execution pipe latency) for the current thread is introduced for the thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. If this field is **Atomic**, the next instruction gets highest priority in the thread arbitration for the execution pipelines.

Field	Description
	Switch is designed primarily as a safety feature in case there are race conditions for certain instructions. 00b = Normal Thread Control. Execution may or may not be preempted by another thread following this instruction. 01b = **Atomic**. Prevent any thread switch immediately following this instruction. Always execute the next instruction (which may not be next sequentially if the current instruction branches). Atomic can be used with send, sends, sendc, sendsc ONLY. 10b = **Switch**. Force a switch to another thread after this instruction and before the next instruction. 11b = Reserved
UIP	**Update Instruction Pointer.** A Doubleword Signed Integer offset relative to the current IP (which references the current instruction) in units of bytes. Typically an immediate value in the instruction. For the *brc* (Branch Converging) instruction, both JIP and UIP can be contained in a register.

In the following table of operand fields, use just the last part of the field name. For example, to find the Src1.ChanSel field, look for ChanSel.

Source or Destination Operand Fields (Alphabetically by Short Name)

Field	Description
AddrImm	**Address Immediate.** A 10-bit signed integer offset in units of bytes, only used with the Indirect Addressing Mode. In that addressing mode, the Address Immediate value is added to an address subregister value to determine the operand's address in the GRF. Allowed for any GRF register operand, destination or source. ARF registers cannot be accessed with indirect addressing. The Address Immediate field cannot cover the 4K-byte range of the thread's GRF. Whatever address subregister value is used along with this offset provides a window into the GRF, limited by the offset range. In the Align16 Access Mode, the low four bits of AddrImm are zero and do not appear in the instruction format. Format = S9 Range = [-512, 511]
AddrMode	**Addressing Mode.** Whether the destination register and subregister are encoded in the instruction (Direct Addressing) or calculated using the contents of an address subregister and an offset (Indirect Addressing). This field applies to source and destination register operands, for instructions with 1 or 2 source operands. 0 = Direct Addressing ("Direct"). Direct register addressing. 1 = Indirect Addressing ("Register-Indirect" or "Indirect"). Register-indirect register addressing. Instructions with 3 source operands use Direct Addressing.

Field	Description
AddrSubRegNum	**Address Sub-Register Number.** The address register contains 16 Word-sized subregisters. This field is only used with the Indirect Addressing Mode and specifies the address subregister containing a value added to the Address Immediate value to determine the operand address. Format = U4 Range = [0, 15] Some variations of Indirect Addressing use multiple address subregisters, where AddrSubRegNum determines the first subregister used.
ChanEn	**Channel Enable.** Dst.ChanEn, used only for the destination operand and only in the Align16 Access Mode. Provides four channel enable bits applied modulo four to all *ExecSize* channels. For example, 0xF enables all channels, 0 disables all channels, 0xA enables odd-numbered channels, and so on. The assembler mnemonics are **x**, **y**, **z**, and **w** for channels 0, 1, 2, and 3 respectively. If MaskCtrl is 1 (mnemonic **NoMask**) then all channels are enabled regardless of the ChanEn value, equivalent to ChanEn of 0xF (**xyzw**). Predication and execution masking, in addition to ChanEn and MaskCtrl, determine what channels are actually written. For i in 0, 1, 2, 3: Bit i = 0 For channel j where j % 4 == i, disable writing that channel. Bit i = 1 For channel j where j % 4 == i, enable writing that channel.
ChanSel	**Channel Select.** This field controls the channel swizzle for a non-immediate source operand in the Align16 access mode. It is not used for immediate operands, destination operands, or in the Align1 access mode. The normally sequential channel assignment can be altered by explicitly identifying neighboring data elements for each channel. Out of the 8-bit field, 2 bits are assigned for each channel within the group of 4. ChanSel[1:0], [3:2], [5:4] and [7:6] are for channel 0 (**x**), 1 (**y**), 2 (**z**), and 3 (**w**) in the group, respectively. When operating on 64-bit operands, these channel selects must be used in pairs to select a contiguous 64-bit source. For example with an execution size of 8, r0.0<4>.zywz:f assigns the channels as follows: Chan0 = Data2, Chan1 = Data1, Chan2 = Data3, Chan3 = Data2; Chan4 = Data6, Chan5 = Data5, Chan6 = Data7, Chan7 = Data6. The 2-bit Channel Selection field for each channel within the group of 4 is defined as: 00b = **x**. Channel 0 is selected for the corresponding execution channel. 01b = **y**. Channel 1 is selected for the corresponding execution channel. 10b = **z**. Channel 2 is selected for the corresponding execution channel. 11b = **w**. Channel 3 is selected for the corresponding execution channel. Note: When using channel select for 64-bit operands, the valid selects are **.xy** and **.zw**. This is required to pick a pair of DWords.
Desc	**Message descriptor.** A 31 bit immediate message descriptor for *send*, *sendc*, *sends* and *sendsc* instructions.This field is not used for other instructions.
DstType	**Destination Type.** Dst.DstType specifies the numeric data type of the destination operand dst. The bits of the destination operand are interpreted as the identified numeric data type, rather than coerced into a type implied by the operator. For a *send* or *sendc* instruction, this field applies to CurrDst, the current destination operand. Three source instructions use a 3-bit encoding that allows fewer data types.

Field	Description
ExDesc	**Extended message descriptor.** A 32 bit immediate extended message descriptor for *send*, *sendc*, *sends* and *sendsc* instructions.This field is not used for other instructions.
HorzStride	**Horizontal Stride.** Is the distance in units of data element size between two adjacent data elements within a row (horizontal) in the register region for an operand. A horizontal stride of 0 is used for a row that is one-element wide, useful when an instruction repeats a column value or repeats a scalar value. For example, adding a single column to every column in a 2D array or adding a scalar to every element in a 2D array uses HorzStride of 0. A horizontal stride of 1 indicates that elements are adjacent within a row. References to HorzStride in this volume normally reference the value not the encoding, so there are references to HorzStride of 4, which is encoded as 11b. This field applies to both source and destination register operands. This field is used with both direct and indirect addressing. 00b = 0 Elements 01b = 1 Element 10b = 2 Elements 11b = 4 Elements See the Register Region Restrictions section for rules that constrain *HorzStride* in relation to other region parameters.
Imm[28:0]	A 29-bit immediate message descriptor for a *send* or *sendc* instruction. This field is not used for other instructions.
Imm32	A 32-bit immediate data field for an immediate source operand. Only one source operand can be immediate, the last source operand. Of course a destination operand is never immediate. For a two-source instruction, src1 can be immediate; for a one-source instruction src0 can be. The source type for an immediate operand cannot be B or UB (signed or unsigned byte). The source type can be one of the packed vector types that are only allowed as immediate operands: V, UV, or VF. For the W or UW (signed or unsigned word) source types, the 16-bit value must be replicated in both the low and high words of the 32-bit immediate value. The low order bits are directly used when fewer than 32-bits are needed for the source type. The 32-bit value is not coerced into the designated type. See the Numeric Data Types section for information about data types and their ranges. In *send*, *sendc*, *sends* or *sendsc* instruction, refers to the 31bit immediate message descriptor field and the 32bit immediate extended message descriptor field.
Imm64	A 64-bit immediate data field for an immediate source operand, only used for a one-source instruction with a 64-bit Source Type (DF, Q, or UQ).
RegFile	**Register File.** Select a source or destination register file or indicate an immediate source operand: 00b = ARF Architecture Register File. Only allowed for src0 or destination. 01b = GRF General Register File. Allowed for any source or destination. 10b = Reserved. 11b = Immediate operand. Only allowed for the last source operand. Not allowed for the

Field	Description
	destination operand or for any other source operand. Note that for flow control instructions requiring two offsets, regfile of source0 is required to be immediate since the 64b for immediates occupy the DW2 and DW3.
RegNum	**Register Number.** The register number for the operand. For a GRF register, is the part of a register address that aligns to a 256-bit (32-byte) boundary. For an ARF register, this field is encoded such that MSBs identify the architecture register type and LSBs provide the register number. An ARF register can only be dst or src0. Any src1 or src2 operands cannot be ARF registers. RegNum and SubRegNum together provide the byte-aligned address for the origin of a register region. RegNum provides bits 12:5 of that address. For one-source and two-source instructions, SubregNum provides bits 4:0. For three-source instructions, the address must be DWord-aligned; SubRegNum provides bits 4:2 of the address and bits 1:0 are zero. This field is present for the direct addressing mode and not present for indirect addressing. This field applies to both source and destination operands. Format = U8, if RegFile = GRF. 0x00 to 0x7F = Register number in the range [0, 127]. 0x80 to 0xFF = Reserved. Format = 8-bit encoding, if RegFile = ARF. This field encodes the architecture register type as well as providing the register number. See the ARF Registers Overview section and the sections for individual ARF registers for details.
RepCtrl	**Replicate Control.** This field is only present in three-source instructions, for each of the three source operands. It controls replication of the starting channel to all channels in the execution size. ChanSel does not apply when Replicate Control is set. This is applicable to 32b datatypes and 16b datatype. 64b datatypes cannot use the replicate control. 0 = No replication. 1 = Replicate across all channels.
SelReg32Desc	**Select Reg32 for message descriptor.** In *sends* or *sendsc* instruction, refers to the selection of Reg32 for the message descriptor field. 0 = Desc. 1 = Reg32. First Dword of Address Register is used for message descriptor.
SpecialAcc	**Special Accumulator.** Specify the accumulator numbers used by the IEEE macro instructions. The 8 special accumulators, acc2 to acc9 are encoded consecutively from 0000b to 0111b and noacc, indicating no special accumulator used is encoded as 1000b.
SrcMod	**Source Modifier.** Specify any numeric (normally) or logical (for logic instructions) modification to a source value before delivery to the execution pipe. The numeric value of each data element of a source operand can optionally have its absolute value taken, its sign inverted (arithmetic negation), or both (absolute value followed by arithmetic negation producing a guaranteed negative value). When used with logic instructions (*and*, *not*, *or*, *xor*), this field indicates whether the source bits are inverted (bitwise NOT) before delivery to the execution pipe, regardless of the source type. This field only applies to source operands. It does not apply to the destination. This field is not present for an immediate source operand.

Field	Description
	For no modification, there is no assembler notation or syntax. Encoding for all instructions other than logic instructions: 00b = No modification (normal). 01b = **(abs)**. Absolute value. 10b = **-**. Negation. 11b = **-(abs)**. Negation of the absolute value (forced negative value). Encoding for logic instructions: 00b = No modification (normal). 01b = No modification (normal). This encoding cannot be selected in the assembler syntax. 10b = **~** Indicates a bitwise NOT, inverting the source bits. 11b = No modification (normal). This encoding cannot be selected in the assembler syntax.
SrcType	**Source Type.** Specifies the numeric data type of a source operand. In a two-source instruction, each source operand has its own source type field. In a three-source instruction, one source type is used for all three source operands. The bits of a source operand are interpreted as the identified numeric data type, rather than coerced into a type implied by the operator. Depending on the RegFile field for the source, this field uses one of two encodings. For a non-immediate source (from a register file), use the Source Register Type Encoding, which is identical to the Destination Type encoding. For an immediate source, use the Source Immediate Type Encoding, which does not support signed or unsigned byte immediate values and does support the three packed vector types, V, UV, and VF. Note that three-source instructions do not support immediate operands, that only the second source (src1) of a two-source instruction can be immediate, and that 64-bit immediate values (DF, Q, or UQ) can only be used with one-source instructions. In a two-source instruction with a V (Packed Signed Half-Byte Integer Vector) or UV (Packed Unsigned Half-Byte Integer Vector) immediate operand, the other source operand must have a type compatible with packed word execution mode, one of B, UB, W, or UW. Note that DF (Double Float) and HF (Half Float) have different encodings in the Source Regster Type Encoding and the Source Immediate Type Encoding. The Source Register Type Encoding and Source Immediate Type Encoding lists apply to instructions with one or two source operands. Three source instructions can use operands with mixed-mode precision. When SrcType field is set to :f or :hf it defines precision for source 0 only, and fields Src1Type and Src2Type define precision for other source operands: 0b = **:f**. Single precision Float (32-bit). 1b = **:hf**. Half precision Float (16-bit).

Field	Description
SubRegNum	**Sub-Register Number.** The subregister number for the operand. For a GRF register, is the byte address within a 256-bit (32-byte) register. For an ARF register, determines the sub-register number according to the specified encoding for the given architecture register. RegNum and SubRegNum together provide the byte-aligned address for the origin of a GRF register region. RegNum provides bits 12:5 of that address. For one-source and two-source instructions, SubregNum provides bits 4:0. For three-source instructions, the address must be DWord-aligned; SubRegNum provides bits 4:2 of the address and bits 1:0 are zero. **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. This field is present for the direct addressing mode and not present for indirect addressing. This field applies to both source and destination operands. Format = U5, if RegFile = GRF and the instruction has fewer than three source operands. 0x00 to 0x1F = Sub-Register number in the range of [0, 31]. Format = U3, if RegFile = GRF and the instruction has three source operands. 0x0 to 0x7 = Sub-Register number MSBs in the range of [0,7]. The two LSBs are zero. Format = U4, if RegFile = GRF and the instruction has three source operands. 0x0 to 0x15 = Sub-Register number MSBs in the range of [0,15]. Format = 5-bit encoding, if RegFile = ARF. See the ARF Registers Overview section and the sections for individual ARF registers for details.
VertStride	**Vertical Stride.** The vertical stride of a source operand's register region in units of data element size. Supported values are 0, powers of 2 from 1 to 32, and a special encoding used for indirect addressing in Align1 mode. Values greater than 32 are not supported due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total). The special encoding 1111b (0xF) is only valid when the operand is in register-indirect addressing mode (AddrMode = 1). If this field is set to 0xF, one or more sub-registers of the address registers may be used to compute source addresses. Each address sub-register provides the origin for a row of data elements. The number of address sub-registers used is equal to instruction's ExecSize / source operand's Width. This field only applies to source operands. It does not apply to the destination. For Align16 access mode, only encodings of 0000b, 0010b, and 0011b are allowed. Other codes are reserved. Note 1: A Vertical Stride larger than 32 is not allowed due to the restriction that a source

Field	Description
	operand must reside within two adjacent 256-bit registers (64 bytes total). Note 2: In Align16 access mode, as encoding 1111b is reserved, only single-index indirect addressing is supported. Note 3: If indirect addressing is supported for src1, the encoding 1111b is reserved for src1 and only single-index indirect addressing is supported. Note 4: The encoding 0010b is used for 64-bit operands (types DF, Q, or UQ). 0000b = 0 Elements 0001b = 1 Element. Align1 mode only. 0010b = 2 Elements 0011b = 4 Elements 0100b = 8 Elements. Align1 mode only. 0101b = 16 Elements (applies to byte or word operands only). Align1 mode only. 0110b = 32 Elements (applies to byte operands only). Align1 mode only. 0111b-1110b = Reserved. 1111b = VxH or Vx1 mode (only valid for register-indirect addressing in Align1 mode).
Width	**Width.** The number of elements in the horizontal dimension of the region for a source operand. This field cannot exceed the ExecSize field of the instruction. This field only applies to source operands. It does not apply to the destination. 000b = 1 Element 001b = 2 Elements 010b = 4 Elements 011b = 8 Elements 100b = 16 Elements 101b-111b = Reserved Note that with ExecSize of 32, because the maximum Width is 16, there are at least two rows in a source region.

Native Instruction Layouts

This section describes the Execution Unit instruction formats.

DWord 0, bits 31:0 of the 128-bit instruction, has the same format regardless of the number of source operands.

The following three tables cover the most common instruction format, for instructions with 1 or 2 source operands; then the format for the few instructions with 3 source operands; and finally format variations used by a few specific instructions, including branching instructions.

Execution Unit Instruction Format for 1 or 2 Source Operands

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
Any Imm32 32-bit or Imm64 64-bit immediate operand uses bits 127:96, replacing the following fields.					
127:122	Reserved				
121	Varies based on AddrMode	Reserved		Src1.AddrImm[9]	
120:117	Src1.VertStride				
116	Varies based on AccessMode	Reserved	Src1.Width	Reserved	Src1.Width
115:114		Src1.ChanSel[7:4]		Src1.ChanSel[7:4]	
113:112			Src1.HorzStride		Src1.HorzStride
111	Src1.AddrMode				
110:109	Src1.SrcMod				
108:105	Src1.SrcMod	Src1.RegNum		Src1.AddrSubRegNum	
104:101				Src1.AddrImm[8:4]	Src1.AddrImm[8:0]
100		Src1.SubRegNum[4]	Src1.SubRegNum[4:0]		
99:96		Src1.ChanSel[3:0]		Src1.ChanSel[3:0]	
Any Imm64 64-bit immediate operand uses bits 127:64, replacing the following fields.					
95	Varies based on AddrMode	Reserved		Src0.AddrImm[9]	
94:91	Src1.SrcType				
90:89	Src1.RegFile				
88:85	Src0.VertStride				
84	Varies based on AccessMode	Reserved	Src0.Width	Reserved	Src0.Width
83:82		Src0.ChanSel[7:4]		Src0.ChanSel[7:4]	
81:80			Src0.HorzStride		Src0.HorzStride
79	Src0.AddrMode				
78:77	Src0.SrcMod				
76:73	Varies based on AddrMode and AccessMode	Src0.RegNum		Src0.AddrSubRegNum	
72:69				Src0.AddrImm[8:4]	Src0.AddrImm[8:0]
68		Src0.SubRegNum[4]	Src0.SubRegNum[4:0]		
67:64		Src0.ChanSel[3:0]		Src0.ChanSel[3:0]	
63	Dst.AddrMode				

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
62:61	Varies based on AccessMode	Reserved	Dst.HorzStride	Reserved	Dst.HorzStride
60:57	Varies based on AddrMode and AccessMode	Dst.RegNum		Dst.AddrSubRegNum	
56:53				Dst.AddrImm[8:4]	Dst.AddrImm[8:0]
52		Dst.SubRegNum[4]	Dst.SubRegNum[4:0]		
51:48		Dst.ChanEn[3:0]		Dst.ChanEn[3:0]	
47	Varies based on AddrMode	Reserved : MBZ		Dst.AddrImm[9]	
46:43	Src0.SrcType				
42:41	Src0.RegFile				
40:37	Dst.DstType				
36:35	Dst.RegFile				
34	MaskCtrl				
33:32	FlagRegNum / FlagSubRegNum				
31	Saturate				
29	CmptCtrl				
28	AccWrCtrl/BranchCtrl				
27:24	CondModifier				
23:21	ExecSize				
20	PredInv				
19:16	PredCtrl				
15:14	ThreadCtrl				
13:12	QtrCtrl				
11	NibCtrl				
10:9	DepCtrl				
8	AccessMode				
7	Reserved (for future Opcode expansion)				
6:0	Opcode				

The 3-source operand instructions are:

1. *bfe* - Bit Field Extract
2. *bfi2* - Bit Field Insert 2
3. *lrp* - Linear Interpolation
4. *mad* - Multiply Add
5. *madm* - Multiply Add for Macro

In the 3-source instruction format, the upper QWord contains three groups of 21 bits for the three source operands, where each group contains four fields in 20 bits and otherwise adjacent groups are separated by single reserved bits.

Specific instructions have different instruction formats as described below. These instructions include send / sendc, math, and branch instructions.

Execution Unit Instruction Format for Specific Instructions

The instruction fields Src0.VertStride, Src0.Width[1:0], Src0.ChanSel[7:4], Src0.HorzStride, Src0.ChanSel[3:0], Src0.AddrImm[3:0], Src1.SrcType and AccWrCtrl are not used for send/sendc/sends/sendsc instructions. Additionally for sends/sendsc instructions Src0.SrcMod, Dst.HorzStride, Dst.ChanEn, Dst.AddrImm[3:0] and Src0.SrcType are not used.

Src0.RegFile[1], Src1.RegFile[1] and Dst.RegFile[1] are implicitly set to 0, and Src0.RegFile[0] is implicitly set as 1 for sends/sendsc instructions. Also note, ExDesc[4] and ExDesc[16:9] are unused and not encoded below.

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description			
		send / sendc	sends / sendsc	math	Branch Instructions
127	Reserved	ExDesc[5] (EOT)			JIP[31:0]
126:125	Reserved	Desc[30:0] / Reg32			
124:121	Reserved				
120:117	Src1.VertStride				
116:112	Varies based on AccessMode				
111	Src1.AddrMode				
110:109	Src1.SrcMod				
108:96	Varies based on AddrMode and AccessMode				
95	Reserved		ExDesc[31:16]/Reg32		UIP[31:0] (2-offset branches)
94:91	Src1.SrcType	ExDesc[31:28]			
90:89	Src1.RegFile				
88:85	Src0.VertStride	ExDesc[27:24]			
84	Varies based on AccessMode				
83:80	Varies based on AccessMode	ExDesc[23:20]			
79	Src0.AddrMode				
78	Src0.SrcMod[1]		Src0.AddrImm[9]		
77	Src0.SrcMod[0]		SelReg32Desc		
76:68	Varies based on AddrMode and AccessMode				
67:64	Varies based on AddrMode and AccessMode	ExDesc[19:16]	ExDesc[9:6]		

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description			
		send / sendc	sends / sendsc	math	Branch Instructions
63	Dst.AddrMode				
62	Varies based on AccessMode		Dst.AddrImm[9]		
61	Varies based on AccessMode		SelReg32ExDesc		
60:52	Varies based on AddrMode and AccessMode				
51:48	Varies based on AddrMode and AccessMode		Src1.RegNum[7:4]		
47	Reserved		Src1.RegNum[3:0]		
46:44	Src0.SrcType				
43	Src0.SrcType				
42	Src0.RegFile[1]				
41	Src0.RegFile[0]				
40:37	Dst.DstType				
36	Dst.RegFile[1]		Src1.RegFile[0]		
35	Dst.RegFile[0]				
34	MaskCtrl				
33:32	FlagRegNum / FlagSubRegNum				
31	Saturate				
30					
29	CmptCtrl				
28	AccWrCtrl	NoSrcDepSet			
27:24	CondModifier	ExDesc[3:0] (SFID)		FC[3:0]	MBZ
23:21	ExecSize				
20	PredInv				
19:16	PredCtrl				
15:14	ThreadCtrl				
13:12	QtrCtrl				
11	NibCtrl				
10:9	DepCtrl				
8	AccessMode				
7	Reserved (for future Opcode expansion)				
6:0	Opcode				

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.

Description
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.

Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

The opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxb are reserved.

Note: Opcodes appear in the overall Instruction Set Summary Table as well. The following subsections still serve the purpose of describing various instruction groups.

Move and Logic Instructions

This instruction group has an opcode format of 00xxxxxb.

- The opcodes for move instructions (*mov*, *sel* and *movi*) share the common 5 MSBs in the form of 00000xxb.
- The opcodes for logic instructions (*not*, *and*, *or*, and *xor*) share the common 5 MSBs in the form of 00001xxb.
- The opcodes for shift instructions (*shr, shl,* ***asr***, **ror**, and **rol**) share the common 4 MSBs in the form of 0001xxxb.
- The opcodes for compare instructions (*cmp* and *cmpn*) share the common 6 MSBs in the form of 001000xb. 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
dec	hex				
1	0x01	mov	Component-wise move	1	1
2	0x02	sel	Component-wise selective move based on predication	2	1
3	0x03	movi	Fast component-wise indexed move	1	1
4	0x04	not	Component-wise one's complement (bitwise not)	1	1
5	0x05	and	Component-wise logical AND (bitwise and)	2	1
6	0x06	or	Component-wise logical OR (bitwise or)	2	1
7	0x07	xor	Component-wise logical XOR (bitwise xor)	2	1
8	0x08	shr	Component-wise logical shift right	2	1
9	0x09	shl	Component-wise logical shift left	2	1
10	0x0A	smov	Scattered Move	1	1
11	0x0B	*Reserved*			
12	0x0C	asr	Component-wise arithmetic shift right	2	1
13	0x0D	*Reserved*			
16	0x10	cmp	Component-wise compare, store condition code in destination	2	1
17	0x11	cmpn	Component-wise compare-NaN, store condition code in destination	2	1
18	0x12	csel	Component-wise selective move based on result of compare	3	1
19	0x13	*Reserved*			
20	0x14	*Reserved*			
21	0x15	*Reserved*			
22	0x16	*Reserved*			
23	0x17	bfrev	Reverse bits	1	1
24	0x18	bfe	Bitfield extract	3	1
25	0x19	bfi1	Bitfield insert macro instruction 1, generate mask	2	1
26	0x1A	bfi2	Bitfield insert macro instruction 2, insert based on mask	3	1
27-31	0x1B-0x1F	*Reserved*			

Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	jmpi	Jump indexed	1	0

Opcode		Instruction	Description	#src	#dst
dec	hex				
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	*Reserved*			
39	0x27	*Reserved*			
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				
48	0x30	wait	Wait for (external) notification	1	0
49	0x31	send	Send	1	1
50	0x32	sendc	Conditional Send (based on TDR)	1	1
51	0x33	sends	Split Send	2	1
52	0x34	sendsc	Conditional Split Send (based on TDR)	2	1
53-55	0x35-0x37	*Reserved*			
56	0x38	math	Math functions for extended math pipeline	1/2	1/2
57-63	0x39-0x3F	*Reserved*			

Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

Parallel Arithmetic Instructions

Opcode		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				
80	0x50	sad2	2-wide sum of absolute difference	2	1
81	0x51	sada2	2-wide sad accumulate	2	1
82-83	0x52-0x53	*Reserved*			
84	0x54	dp4	4-wide dot product for 4-vector	2	1
85	0x55	dph	4-wide homogenous dot product for 4-vector	2	1
86	0x56	dp3	3-wide dot product for 4-vector	2	1
87	0x57	dp2	2-wide dot product for 4-vector	2	1
88	0x58	*Reserved*			
89	0x59	line	Component-wise line equation computation (a multiply-add)	2	1
90	0x5A	pln	Component-wise floating point plane equation computation (a	2	1

Opcode		Instruction	Description	#src	#dst
dec	hex				
			multiply-multiply-add)		
91	0x5B	fma(mad)	Component-wise floating point mad computation (a multiple-add)	3	1
92	0x5C	lrp	Component-wise floating point lrp computation (blend)	3	1
93	0x5D	***fmam(madm)***	Component-wise floating point fused multiply and add for macro operations.	3	1
94-95	0x5E-0x5F	*Reserved*			

Special Instructions

There are two special instructions, namely, *nop* (opcode = 0x7E) and *illegal* (opcode = 0x00).

- *Nop* instruction may be used for instruction padding in memory between two normal instructions to force alignment or to introduce instruction execution delay. Currently, there is no need for between-instruction padding.
- *Illegal* instruction may be used for instruction padding in memory outside the normal instruction sequence such as before or after the kernel program as well as between subroutines.
- *Nop* and *illegal* instructions do not have source operands or destination operand. Therefore, they do not implicitly update the accumulator register. They cannot be compressed.

Special Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
0	0x00	illegal	Illegal instruction	0	0
96-125	0x60-0x7D	*Reserved*			
126	0x7E	nop	No-op	0	0
127	0x7F	*Reserved*			

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>

|<BinaryAccInstruction>

|<BinaryInstruction>

|<TriInstruction>

|<JumpInstruction>

|<BranchLoopInstruction>

|<ElseInstruction>

|<BreakInstruction>

|<MaskControlInstruction

|<TriInstruction2>

|<CallInstruction>

|<BranchConvIntruction>

|<BranchDivInstruction>

|<MathInstruction>

|<SyncInstruction>

|<SpecialInstruction>

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

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

<UnaryOp>::= "*mov*" | "*frc*" | "*rndu*" | "*rndd*" | "*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**"" | "**bfi1**"" | "**addc**"" | "**subb**"

<BinaryAccInstruction>::= <Predicate> <BinaryAccInst> <ExecSize> dst <SrcAcc> <SrcImm> <InstrOptions>

<BinaryAccInst>::= <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”

<BrancConvInstruction> ::= <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>::= “**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

<SrcAccImm>::=<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**"..."**2^{32}-1**" | "-**2^{31}**"..."**2^{31}-1**"

<Imm16>::="**0**"..."**2^{16}-1**" | "-**2^{15}**"..."**2^{15}-1**"

<ImmDesc>::="**0**"..."**2^{32}-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 compressable.

Instruction Set Summary Tables

The columns in the following tables specify instruction mnemonics, hex opcodes, full names, instruction groups, processor generation (where blank means available for SNB+), 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.

With a dozen columns in these tables, some terse notation is used, like IVB+ for IVB+ in the Project column. If the Project column is blank, an instruction is supported for SNB+, all generations implemented or designed so far, from Sandy Bridge forward.

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 abolute 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?
add	40	Addition	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
addc	4E	Integer Addition with Carry	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
and	05	Logic And	Move and Logic	2	Y	**	Int	Int	N	Equality only
asr	0C	Arithmetic Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
avg	42	Average	Parallel Arithmetic	2	Y	Y	B, UB W, UW D, UD	B, UB W, UW D, UD	Y	Y
bfe	18	Bit Field Extract	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfi1	19	Bit Field Insert 1	Move and Logic	2	Y	N	UD, D	UD, D	N	N

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
bfi2	1A	Bit Field Insert 2	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfrev	17	Bit Field Reverse	Move and Logic	1	Y	N	UD	UD	N	N
brc	23	Branch Converging	Flow Control	0 or 1	Y	N	D		N	N
brd	21	Branch Diverging	Flow Control	0 or 1	Y	N	D		N	N
break	28	Break	Flow Control	0	Y	N			N	N

Instruction Set Summary Table C to E (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
call	2C	Call	Flow Control	0	Y	N		D, UD	N	N
calla	2B	Call Absolute	Flow Control	0	Y	N		D, UD	N	N
cbit	4D	Count Bits Set	Move and Logic	1	Y	N	UB, UW, UD	UD	N	N
cmp	10	Compare	Move and Logic	2	Y	Y	*	*	N	Y
cmpn	11	Compare NaN	Move and Logic	2	Y	Y	*	*	N	Y
cont	29	Continue	Flow Control	0	Y	N			N	N
csel	12	Conditional Select	Move and Logic	3	N	Y	F	F	Y	Y
dp2	57	Dot Product 2	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dp3	56	Dot Product 3	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dp4	54	Dot Product 4	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dph	55	Dot Product Homogeneous	Vector Arithmetic	2	Y	Y	F	F	Y	Y
else	24	Else	Flow Control	0	N	N			N	N
endif	25	End If	Flow Control	0	N	N			N	N

Instruction Set Summary Table F to L (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
fbh	4B	Find First Bit from MSB Side	Move and Logic	1	Y	N	D, UD	UD	N	N
fbl	4C	Find First Bit from LSB Side	Move and Logic	1	Y	N	UD	UD	N	N
frc	43	Fraction	Parallel Arithmetic	1	Y	Y	F	F	N	Y
goto	2E	Goto	Flow Control	0	Y	N			N	N
halt	2A	Halt	Flow Control	0	Y	N			N	N
if	22	If	Flow Control	0	Y	N			N	N
illegal	00	Illegal	Special	0	N	N			N	N
jmpi	20	Jump Indexed	Flow Control	1	Y	N	D		N	N
join	2F	Join	Flow Control	0	Y	N			N	N
line	59	Line	Vector Arithmetic	2	Y	Y	F	F	Y	Y
lrp	5C	Linear Interpolation	Vector Arithmetic	3	Y	Y	F	F	N	Y
lzd	4A	Leading Zero Detection	Move and Logic	1	Y	Y	D, UD	UD	Y	Y

Instruction Set Summary Table M to P (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
mac	48	Multiply Accumulate	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mach	49	Multiply Accumulate High	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mad	5B	Multiply Add	Vector Arithmetic	3	Y	Y	*	*	Y	Y
madm	5D	Multiply Add for Macro	Vector Arithmetic	3	Y	Y	*	*	Y	Y

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
math	38	Extended Math Function	Miscellaneous	2	Y	N	*	*	Y	N
mov	01	Move	Move and Logic	1	Y	Y	*	*	Y	Y
movi	03	Move Indexed	Move and Logic	1	Y	Y	*	*	Y	N
mul	41	Multiply	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
nop	7E	No Operation	Special	0	N	N			N	N
not	04	Logic Not	Move and Logic	1	Y	**	Int	Int	N	Equality only
or	06	Logic Or	Move and Logic	2	Y	**	Int	Int	N	Equality only
pln	5A	Plane	Vector Arithmetic	2	Y	Y	F	F	Y	Y

Instruction Set Summary Table R to X (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
ret	2D	Return	Flow Control	1	Y	N	D, UD		N	N
rndd	45	Round Down	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rnde	46	Round to Nearest or Even	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndu	44	Round Up	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndz	47	Round to Zero	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
sad2	50	Sum of Absolute Difference 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sada2	51	Sum of Absolute Difference Accumulate 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sel	02	Select	Move and Logic	2	Y	Y	*	*	Y	Y
send	31	Send Message	Miscellaneous	1	Y	N	*	*	N	N
sendc	32	Conditional Send Message	Miscellaneous	1	Y	N	*	*	N	N
sends	33	Split Send Message	Miscellaneous	2	Y	N	*	*	N	N
sendsc	34	Conditional Split Send Message	Miscellaneous	2	Y	N	*	*	N	N
shl	09	Shift Left	Move and Logic	2	Y	Y	Int	Int	Y	Y
shr	08	Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
smov	0A	Scattered Move	Move and Logic	1	Y	N	*	*	N	N
subb	4F	Integer Subtraction with Borrow	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
wait	30	Wait	Miscellaneous	1	N	N	UD	UD	N	N
while	27	While	Flow Control	0	Y	N			N	N
xor	07	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 use the accumulator as an implicit source value cannot specify an explicit accumulator source operand.
3. Instructions that specify an implicit accumulator destination (with AccWrEn) cannot specify an explicit accumulator destination operand.
4. An instruction with both an explicit accumulator source operand and an explicit accumulator destination operand must specify the same accumulator register as the source and the destination.

Description
5. Instructions with three source operands cannot use explicit accumulator operands. AccWrEn may be allowed for implicitly updating the accumulator.

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
add	No restrictions.
addc	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
and	Source modifier is not allowed if source is an accumulator.
asr *avg*	No restrictions.
bfe	No accumulator access, implicit or explicit.

Instructions	Accumulator Restrictions
bfi1 *bfi2* *bfrev* *cbit*	
cmp	No restrictions.
cmpn	No restrictions.
csel	No restrictions.
dp2 *dp3* *dp4* *dph*	Source operands cannot be accumulators.
fbh *fbl*	No accumulator access, implicit or explicit.
frc	No restrictions.
line	Source operands cannot be accumulators.
lrp	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
lzd	No restrictions.
mac	Accumulator is an implicit source and thus cannot be an explicit source operand.
mach	Accumulator is an implicit source and thus cannot be an explicit source operand. AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
madm	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
math	No accumulator access, implicit or explicit.
mov	An accumulator can be a source or destination operand but not both.
movi	Source operands cannot be accumulators.
mul	Integer source operands cannot be accumulators.
not *or*	Source modifier is not allowed if source is an accumulator.
pln	Source operands cannot be accumulators.
rndd	No restrictions.

Instructions	Accumulator Restrictions
rnde *rndu* *rndz*	
sad2 *sada2*	Source operands cannot be accumulators.
sel	No restrictions.
shl	Accumulator cannot be destination, implicit or explicit.
shr	No restrictions.
smov	No restrictions.
subb	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
xor	Source modifier is not allowed if source is an accumulator.

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 GEN assembly instruction format.

```
[(pred)] opcode (exec_size) dst src0 [src1]
```

Square brackets "[ ]" indicate that a field is optional. Saturation modifiers and instruction options are omitted for simplicity.

General Macros and Definitions

INST_MIN_SIZE is defined as a constant of 8 bytes.

```
#define INST_MIN_SIZE  8  // Instruction minimum size in bytes (for the compact instruction format)
```

The floor function converts a floating point value to an integral floating point value. For a given floating point value, from its closest two integral float values, floor returns the one that is closer to negative infinity. For example, floor(1.3f) = 1.0f and floor(-1.3f) = -2.0f.

```
float floor(float g)
{
    return maximum(any integral float f: f <= g)
}
```

The Condition function takes the conditional signals {SN, ZR, OF, IN, NC} of result, generates a Boolean value according to a conditional evaluation controlled by the conditional modifier cmod, and returns the Boolean.

```
Bool Condition(result, cmod)
```

The ConditionNaN function takes the conditional signals {SN, ZR, OF, IN, NC, NS} of result, generates a Boolean value according to a conditional evaluation controlled by the conditional modifier cmod, and returns the Boolean. The only difference between Condition and ConditionNaN is that ConditionNaN uses the NS (NaN of the second source) signal.

```
Bool ConditionNaN(result, cmod)
```

The Jump function jumps the instruction sequence from the current instruction location by InstCount 8-byte units, where each 16-byte native instruction is two units and each 8-byte compact instruction is one unit. If InstCount is positive and greater than zero, is an unconditional jump forward. If InstCount is negative, is an unconditional jump backward. If InstCount is zero, IP stays on the current instruction in an infinite loop.

```
void Jump(int InstCount)
{
    IP = IP + (InstCount * INST_MIN_SIZE)
}
```

Evaluate Write Enable

The WrEn should be evaluated as below.

Note: MaskCtrl = NoMask (1) skips the check for PcIP[n] == ExIP before enabling a channel.

```
if ( MaskCtrl == 1 ) {
  for ( n = 0; n < exec_size; n++ ) {
    WrEn[n] = 1;
  }
}
else {
  for ( n = 0; n < exec_size; n++ ) {
    if ( PcIP[n] == ExIP ) {
      WrEn[n] = 1;
    }
    else {
      WrEn[n] = 0;
    }
  }
}

if ( PredCtrl != 0000b ) {
  for ( n = 0; n < exec_size; n++ ) {
    WrEn[n] = WrEn[n] & PMask[n];
  }
}

for ( n = exec_size; n < 32; n++ ) {
  WrEn[n] = 0;
}
```

EUISA Instructions

Symbol	Name	Source
add	**Addition**	EUISA
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
break	**Break**	EUISA
call	**Call**	EUISA
calla	**Call Absolute**	EUISA
cmp	**Compare**	EUISA
cmpn	**Compare NaN**	EUISA
csel	**Conditional Select**	EUISA
sendc	**Conditional Send Message**	EUISA
sendsc	**Conditional Split Send Message**	EUISA

cont	**Continue**	EUISA
cbit	**Count Bits Set**	EUISA
dp2	**Dot Product 2**	EUISA
dp3	**Dot Product 3**	EUISA
dp4	**Dot Product 4**	EUISA
dph	**Dot Product Homogeneous**	EUISA
else	**Else**	EUISA
endif	**End If**	EUISA
math	**Extended Math Function** • 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
jmpi	**Jump Indexed**	EUISA
lzd	**Leading Zero Detection**	EUISA
line	**Line**	EUISA
lrp	**Linear Interpolation**	EUISA
and	**Logic And**	EUISA
not	**Logic Not**	EUISA
or	**Logic Or**	EUISA
xor	**Logic Xor**	EUISA

mov	**Move**	EUISA
movi	**Move Indexed**	EUISA
mul	**Multiply**	EUISA
mac	**Multiply Accumulate**	EUISA
mach	**Multiply Accumulate High**	EUISA
mad	**Multiply Add**	EUISA
madm	**Multiply Add for Macro**	EUISA
nop	**No Operation**	EUISA
pln	**Plane**	EUISA
ret	**Return**	EUISA
rndd rnde rndu rndz	**Round Instructions** ▪ **Round Down** ▪ **Round to Nearest or Even** ▪ **Round Up** ▪ **Round to Zero**	EUISA
smov	**Scattered Move**	EUISA
sel	**Select**	EUISA
send	**Send Message**	EUISA
shl	**Shift Left**	EUISA
shr	**Shift Right**	EUISA
sends	**Split Send Message**	EUISA
sad	**Sum of Absolute Difference 2**	EUISA
sada2	**Sum of Absolute Difference Accumulate 2**	EUISA
wait	**Wait Notification**	EUISA
while	**While**	EUISA

Round Instructions

rndd - Round Down

rndu - Round Up

rnde - Round to Nearest or Even

rndz - Round to Zero

rndd – Round Down

Description:

The *rndd* instruction takes component-wise floating point downward rounding (to the integral float number closer to negative infinity) of src0 and storing the rounded integral float results in dst. This is commonly referred to as the floor() function.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Down in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	-finite	^	–0	+0	+0	**	+inf	NaN
Notes:									
^	Note: {-1, -0} depending on the Single Precision Denorm Mode.								
**	Result may be {+finite, +0}.								

Floating-Point Round Down in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	-finite	–0	–0	+0	+0	**	+fmax	
Notes:									
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

rnde – Round to Nearest or Even

Description:

The *rnde* instruction takes component-wise floating point round-to-even operation of src0 with results in two pieces – a downward rounded integral float results stored in dst and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in dst to create the final round-to-even values to emulate the round-to-even operation, commonly known as the round() function. The final results are the one of the two integral float values that is nearer to the input values. If 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 src0 is {–inf, +inf, NaN}.								

rndu – Round Up

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of src0, commonly known as the ceiling() function.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Up in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	*	–0	–0	+0	^	+finite	+inf	NaN
Notes:									
*	Result may be {–finite, –0}.								
^	Note: {+1, +0} depending on the Single Precision Denorm Mode.								

Floating-Point Round Up in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	+finite	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

rndz – Round to Zero

Description:

The *rndz* instruction takes component-wise floating point round-to-zero operation of src0 with results in two pieces – a downward rounded integral float results stored in dst and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in dst to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the truncate() function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.

Floating-Point Round to Zero in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	*	–0	–0	+0	+0	**	+inf	NaN
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								

Floating-Point Round to Zero in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	**	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

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
9h	FDIV
Ah	POW
Ch	INT DIV – return quotient only
Dh	INT DIV – return remainder
Eh	INVM
Fh	RSQRTM

INV - Inverse

Precision:1 ULP

Src->	+inf	+0 / +Denorm	- 0 / -Denorm	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN
Dest – ALT mode		+fmax	-fmax		NaN
Src->	+inf	+0	- 0	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN

LOG – Logarithm

Precision:

DirectX 10 and below

If src0 is [0.5..2], **absolute error** must be no more than 2-21. If src0 is (0..0.5) or (2..+INF], **relative error** must be no more than 2-21

Note: In ALT mode log is computed as Log_2 (abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**-inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**-fmax**	**-fmax**		**+F**	**NaN**
Src->	+inf	+0	- 0	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**-inf**	**-inf**	**NaN**	**NaN**	**NaN**

EXP - Exponent

Precision:

DirectX 10 Relative error <= 3 ULP

GPGPU – Relative error <= 4 ULP

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**1**	**1**	**0**	**+F**	**NaN**
Dest – ALT mode		**1**	**1**		**+F**	**NaN**
Src->	+inf	+0	- 0	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**1**	**1**	**0**	**+F**	**NaN**

SQRT - Square Root

Precision:

DirectX 10 (and below) Relative error <= 1 ULP

GPGPU – Relative error <= 4 ULP

Notes:In ALT mode SQRT is computed as SQRT(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**0**	**-0**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**0**	**0**		**+F**	**NaN**
Src->	+inf	**+0**	**-0**	-inf	**-F**	**NaN**
Dest – IEEE mode	+inf	**0**	**-0**	NaN	**NaN**	**NaN**

RSQ - Reciprocal Square Root

Precision:

DirectX 10 and below Relative error <= 3 ULP

GPGPU – Relative error <= 4 ULP

Notes:In ALT mode RSQ is computed as RSQ(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+0**	**+inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**+fmax**	**+fmax**		**+F**	**NaN**
Src->	+inf	+0	-0	-inf	-F	NaN
Dest – IEEE mode	**+0**	**+inf**	**-inf**	**NaN**	**NaN**	**NaN**

POW - Power Function

Precision:

DirectX 10+ do not have precision requirements for POW.

Half_pow precision for GPGPU; i.e., 8192 ULPs with following exception

POWR, POWN is not supported in hardware.

IEEE Mode:

Src0->								
Src1	abs(F > 1)	abs(F < 1)	abs(+F = = 1)	+inf	+0 / +Denorm	-Denorm / -0	-inf	NaN
+inf	+inf	0	NaN	+inf	0	0	+inf	NaN
+0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F			+inf	0	0	NaN	NaN

IEEE Mode:

Src0->								
Src1	abs(F > 1)	abs(F < 1)	abs(+F = = 1)	+inf	+0	-0	-inf	NaN
+inf	+inf	0	NaN	+inf	0	0	+inf	NaN
+0	1	1	1	NaN	NaN	NaN	NaN	NaN
-0	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F			+inf	0	0	NaN	NaN

ALT Mode:

Src0->							
Src1	+F	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
+inf							
+0 / Denorm	1		1	1		1	NaN
-0 / Denorm	1		1	1		1	NaN
-inf							
-F	+F		+fmax	+fmax		+F	NaN
NaN			NaN	NaN		NaN	NaN
+F	+F		0	0		+F	NaN

SIN - SINE

Precision:

DirectX 10 and below Absolute error <= 0.0008 for the range of +/- 32767* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+0	-0		-1 to 1	NaN
Src->	+inf	+0	-0	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN

COS - COSINE

Precision:

DirectX 10 and below Absolute error <= 0.0008 for the range of +/- 32767* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of COS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+1	+1		-1 to 1	NaN
Src->	+inf	+0	-0	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN

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.

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) r1.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 r1.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 r1.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

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.

Workaround
When both srcs are NAN, FDIV produces denorminator NAN as output.

Send Message

send - Send Message

The following information describes the send message.

Send Message

Opcode	Instruction	Description
49 (0x31)	send <dest> <src> <ex_desc> <desc>	Send a message stored in GRF starting at <src> to a shared function identified by <ex_desc> along with control from <desc> with a GRF writeback location at <dest>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Type
Y					[FLT] [INT]

Format:

```
[(pred)] send (exec_size) <dest> <src> <ex_desc> <desc>
```

Syntax:

Syntax
[(pred)] send (exec_size) reg greg imm32 reg32a [(pred)] send (exec_size) reg greg imm32 imm32

Pseudocode:

```
Evaluate(WrEn);
<MsgChEnable> = WrEn;
<SourceReg> = <src>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>, <ex_dest>,
<desc>);
```

Description:

The *send* instruction performs data communication between a thread and external function units, including shared functions (Sampler, Data Port Read, Data Port Write, URB, and Message Gateway) and some fixed functions (e.g. Thread Spawner, who also have an unique Shared Function ID). The *send* instruction adds an entry to the EU's message request queue. The request message is stored in a block of contiguous GRF registers. The response message, if present, will be returned to a block of contiguous GRF registers. The return GRF writes may be in any order depending on the external function units. <src> is the lead GRF register for request. <dest> is the lead GRF register for response. The message descriptor field <desc> contains the Message Length (the number of consecutive GRF registers) and the Response Length (the number of consecutive GRF registers). It also contains the header present bit, and the function control signals. The extend mesage descriptor field <ex_desc> contains the target function ID. WrEn is forwarded to the target function in the message sideband.

The extended message descriptor field <ex_desc> also contains the extended function control field to be sent to the Target Shared Function over message sideband.

The *send* instruction is the only way to terminate a thread. When the EOT (End of Thread) bit of <ex_desc> is set, it indicates the end of thread to the EU, the Thread Dispatcher and, in most cases, the parent fixed function.

Message descriptor field <desc> can be a 32-bit immediate, *imm32*, or a 32-bit scalar register, <reg32a>. GEN restricts that the 32-bit scalar register <reg32a> must be the leading dword of the address register. It should be in the form of a0.0<0;1,0>:ud. When <desc> is a register operand, only the lower 29 bits of <reg32a> are used.

Syntax
<ex_desc> is a 32-bit immediate, imm32. The lower 4bits of the <ex_desc> specifies the SFID for the message. The bit5 of the extended message descriptor, the EOT field, always comes from bit 127 of the instruction word. A thread must terminate with a send instruction with EOT turned on. The higher 16bits, bit31:16 specify the 16bit extended function control field. Interpretation of the extended function control signals is subject to the target external function.

<src> is a 256-bit aligned GRF register. It serves as the leading GRF register of the request.

The source dependency control, {NoSrcDepSet} is used to control the setting of source dependency for the source.

<dest> serves for two purposes: to provide the leading GRF register location for the response message if present, and to provide parameters to form the channel enable sideband signals.

<dest> signals whether there is a response to the message request. It can be either a null register, a direct-addressed GRF register or a register-indirect GRF register. Otherwise, hardware behavior is undefined.

If <dest> is null, there is no response to the request. Meanwhile, the Response Length field in <desc> must be 0. Certain types of message requests, such as memory write (store) through the Data Port, do not want response data from the function unit. If so, the posted destination operand can be null.

If <dest> is a GRF register, the register number is forwarded to the shared function. In this case, the target function unit must send one or more response message phases back to the requesting thread. The number of response message phases must match the Response Length field in <desc>, which of course cannot be zero. For some cases, it could be an empty return message. An empty return message is defined as a single phase message with all channel enables turned off.

The subregister number, horizontal stride, destination mask and type fields of <dest> are always valid and are used in part to generate on the WrEn. This is true even if <dest> is a null register (this is an exception for null as for most cases these fields are ignored by hardware).

The 16-bit channel enables of the message sideband are formed based on the WrEn. Interpretation of the channel enable sideband signals is subject to the target external function. In general for a 'send' instruction with return messages, they are used as the destination dword write mask for the GRF registers starting at <dest>. For a message that has multiple return phases, the same set of channel enable signals applies to all the return phases.

Thread managed memory coherency: A special usage of using non-null <dest> is to support write-commit signaling for memory write service by the Data Port Write unit. If <post_dest> is not null for a memory write request, the Data Port along with the Data Cache or Render Cache will wait until all the posted writes for the request have reached the coherent domain before sending back to the requesting thread an empty message to <dest> register. A memory write reaching the coherent domain, also referred to as reaching the global observable state, means that subsequent read to the same memory location, no matter which thread issues the read, must return the data of the write.

NoDDClr and NoDDChk must not be used for send instruction.

r127 must not be used for return address when there is a src and dest overlap in send instruction.

Message Descriptor Definition

Description
Reserved : MBZ
Data Format. This field specifies the width of data read from sampler or written to render target. Format = U1 0 – Single Precision (32b) 1 – Half Precision (16b)

Reserved : MBZ
Message Length. This field specifies the number of 256-bit GRF registers starting from <src> to be sent out on the request message payload. Valid value ranges from 1 to 15. A value of 0 is considered erroneous. Format = U4 Range = [1,15]
Response Length. This field indicates the number of 256-bit registers expected in the message response. The valid value ranges from 0 to 16. A value 0 indicates that the request message does not expect any response. The largest response supported is 16 GRF registers. Format = U5 Range = [0,16]
Header Present. If set, indicates that the message includes a header. Depending on the target shared function, this field may be restricted to either enabled or disabled. Refer to the specific shared function section for details. Format = Enable
Function Control This field is intended to control the target function unit. Refer to the section on the specific target function unit for details on the contents of this field.

Extended Message Descriptor Definition

Bit	Description
31:16	Extended Function Control. This field is intended to control the target function unit. Refer to the section on the specific target function unit for details on the contents of this field.
15:6	reserved
5	**End Of Thread** This field, if set, indicates that this is the final message of the thread and the thread's resources can be reclaimed.
4	reserved
3:0	**Target Function ID** This field indicates the function unit for which the message is intended. ***Refer to "GPU Overview" document for the mapping of Shared Function IDs***

Restrictions:

Software must obey the following rules in signaling the end of thread using the *send* instruction:

- The posted destination operand must be null.
- No acknowledgement is allowed for the *send* instruction that signifies the end of thread. This is to avoid deadlock as the EU is expecting to free up the terminated thread's resource.
- A thread must terminate with a *send* instruction with message to a shared function on the output message bus; therefore, it cannot terminate with a *send* instruction with message to the following shared functions: Sampler unit, NULL function. For example, a thread may terminate with a URB write message or a render cache write message.
- A root thread originated from the media (generic) pipeline must terminate with a *send* instruction with message to the Thread Spawner unit. A child thread should also terminate with a send to TS. Please refer to the Media Chapter for more detailed description.
- The *send* instruction can not update accumulator registers.
- Saturate is not supported for *send* instruction.
- ThreadCtrl are not supported for *send* instruction.
- The *send* with EOT should use register space R112-R127 for <src>. This is to enable loading of a new thread into the same slot while the message with EOT for current thread is pending dispatch.
- SEND should not use 64-bit datatype for src or dest register.
- When context save and restore of a thread is required, the registers r0-r4 must not be used as *dest or src* registers. This is required to provide a grf register space to save sr1.0 by the system interrupt routine in the event of a mid thread pre-emption.

Extended Message Descriptor Definition

Bits	Description
5	**End Of Thread**This field, if set, indicates that this is the final message of the thread and the thread's resources can be reclaimed.
4	reserved
3:0	**Target Function ID** This field indicates the function unit for which the message is intended. Refer to volume GPU Overview for GPE Function IDs.

EUISA Structures

Name	Source
AddrSubRegNum	Eulsa
DstRegNum	Eulsa
DstSubRegNum	Eulsa

Name	Source
EU_INSTRUCTION_BASIC_ONE_SRC	Eulsa
EU_INSTRUCTION_BASIC_THREE_SRC	Eulsa
EU_INSTRUCTION_BASIC_TWO_SRC	Eulsa
EU_INSTRUCTION_BRANCH_CONDITIONAL	Eulsa
EU_INSTRUCTION_BRANCH_ONE_SRC	Eulsa
EU_INSTRUCTION_BRANCH_TWO_SRC	Eulsa
EU_INSTRUCTION_COMPACT_THREE_SRC	Eulsa
EU_INSTRUCTION_COMPACT_TWO_SRC	Eulsa
EU_INSTRUCTION_CONTROLS	Eulsa
EU_INSTRUCTION_CONTROLS_A	Eulsa
EU_INSTRUCTION_CONTROLS_B	Eulsa
EU_INSTRUCTION_HEADER	Eulsa
EU_INSTRUCTION_ILLEGAL	Eulsa
EU_INSTRUCTION_MATH	Eulsa
EU_INSTRUCTION_NOP	Eulsa
EU_INSTRUCTION_OPERAND_CONTROLS	Eulsa
EU_INSTRUCTION_OPERAND_DST_ALIGN1	Eulsa
EU_INSTRUCTION_OPERAND_DST_ALIGN16	Eulsa
EU_INSTRUCTION_OPERAND_SEND_MSG	Eulsa
EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN1	Eulsa
EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN16	Eulsa
EU_INSTRUCTION_OPERAND_SRC_REG_THREE_SRC	Eulsa
EU_INSTRUCTION_SEND	Eulsa
EU_INSTRUCTION_SENDS	Eulsa
EU_INSTRUCTION_SOURCES_IMM32	Eulsa
EU_INSTRUCTION_SOURCES_REG	Eulsa
EU_INSTRUCTION_SOURCES_REG_IMM	Eulsa
EU_INSTRUCTION_SOURCES_REG_REG	Eulsa
ExtMsgDescpt	Eulsa
FunctionControl	Eulsa
MsgDescpt31	Eulsa
SrcRegNum	Eulsa
SrcSubRegNum	Eulsa

EUISA Enumerations

Name	Source
AddrMode	Eulsa
ChanEn	Eulsa
ChanSel	Eulsa
CondModifier	Eulsa
DepCtrl	Eulsa
DstType	Eulsa
EU_OPCODE	Eulsa
ExecSize	Eulsa
FC	Eulsa
HorzStride	Eulsa
PredCtrl	Eulsa
QtrCtrl	Eulsa
RegFile	Eulsa
RepCtrl	Eulsa
SFID	Eulsa
SrcImmType	Eulsa
SrcIndex	Eulsa
SrcMod	Eulsa
SrcType	Eulsa
ThreadCtrl	Eulsa
VertStride	Eulsa
Width	Eulsa

EU Programming Guide

Assembler Pragmas

Declarations

A register or a register region 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*

For register-indirect-register-addressing mode, the declared symbol can be used to provide immediate address term in the following Cartesian form

<symbol>[IdxReg, RegOff, SubRegOff]<= RegNum (byte-aligned) = [**IdxReg]**+(*RegBase*+ **RegOff**)*32 + (*SubRegBase* + **SubRegOff**)**ElementSize*

or in the following simplified row-aligned form

<symbol>[IdxReg, RegOff]<= RegNum (byte-aligned) = [**IdxReg]**+(*RegBase*+ **RegOff**)*32

or in the form without the immediate address term

<symbol>[IdxReg]<= RegNum (byte-aligned) = [**IdxReg]**+ *RegBase*

Defaults and Defines

The default execution size is set according to the destination register type as the following

Destination Register Type	Default Execution Size
UB | B	(16)
UW | W	(16)
F | UD | D	(8)

The default execution size can be overwritten globally for all instructions using

.**default_execution_size***(Execution_Size)*

or be set according the **destination** register type using

.**default_execution_size**_*Type(Execution_Size)*

The default register type can be set for all register files using

.**default_register_type***Type*

or be set per register file using

.**default_register_type**_*RegFileType*

The default **source** register region for all symbols can be set using

.**default_source_register_region***<VirtStride; Width, HorzStride>*

or be set per register type using

.**default_source_register_region**_*type<VirtStride; Width, HorzStride>*

The default **destination** register region for all symbols can be set using

.**default_destination_register_region***< HorzStride>*

or be set per register type using

.**default_destination_register_region**_*type< HorzStride>*

Finally, the precompiler supports the string replacement statement of .define in the following form

.**define**<symbol>Expression

Programming Note	
Context:	Defaults and Defines
• **.declare** does not support nesting. In other words, each symbol in .declare must be self defined. This would allow the pre-processor to expand all symbols in one pass. • **.define** does support nesting. Only string substitution is supported (currently). • White space within square, angle and round brackets are allowed for easy source code alignment.	

Example Pragma Usages

Example: Declaration for 8x4=32-Byte Regions:

The following symbol Block can be used to address any 8x4 byte region within the Cartisian system of a 16x8 byte GRF register area starting from r0.

Declaration

```
// 32x4 Byte Array.declare BlockBase=r0 ElementSize=1 Region=<32;8,1>Type=b
```

Fully-Expressed Instr

```
mov(32)?:br0.16<32;8,1>:b// r0 xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r1
xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r2 xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r3
xxxxxxxxxxxxxxxxoooooooooxxxxxxxx
```

Short-handed Instr

Mov?:bBlock(0,16)// (0,16): RegNum=0, SubRegNum=16

Example: Declaration for 8x1 Float Regions:

The following symbol *Trans* can be used to address any 8x1 float region within the Cartisian system of a 8x4 float GRF register area starting from r5.

Declaration

```
// 8x4 float Array starting at r5.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f
```

Fully-Expressed Instr

```
mov(8)?:fr6.0<0;8,1>:f// 2nd 16x1 Row of Trans. Matrix // r5 FFFFFFFF// r6 OOOOOOOO// r7
FFFFFFFF// r8 FFFFFFFF
```

Short-handed Instr

mov?:fTrans(1) // RegNum = 5+1 = 6

Example: Declaration for 8x1 Float Regions with 1x1 Indirect Addressing:

Trans region defined (same as in the previous example) is used in conjunction with the address register.

Declaration

```
//8x4 float data array and 16x1 word address array.declare TransBase=r5 ElementSize=4
Region=<0;8,1> Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,224]<0;8,1>:f

Short-handed Instr

mov?:fTrans[a0.0,2] // [a0.0 + 5*32 + 2*32]

Example: Declaration with VxH Indirect Addressing:

The VxH register-indirect-register-addressing for Trans can be provided in the following short-hand form

Declaration

```
//8x4 float data array and word indices.declare TransBase=r5 ElementSize=4 Region=<0;8,1>
Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,224]<1,0>:f

Short-handed Instr

mov?:fTrans[a0.0,2]<1,0> // [a0.0+224] [a0.1+224] ... [a0.7+224]

Example: Declaration with Vx1 Indirect Addressing:

```
As width (4) is smaller than the execution region size (8), multiple indexed registers are
used.
```

Declaration

```
//8x4 float data array and word address array.declare TransBase=r5 ElementSize=4
Region=<0;8,1> Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,244]<4,1>:f

Short-handed Instr

```
mov?:fTrans[a0.0,2]<4,1>// [a0.0+224] [a0.1+224]
```

Assembly Programming Guideline

The following program skeleton illustrates the basic structure of a typical assembly program.

```
                          // single line comment

 /*          block comment
 */

 <preproc_directive>             // macros, include, etc.  Are global – handled by the pre-
processor
 <preproc_directive>             // applies to all code that follows in sequence

 // ------------ some kernel
 .kernel <kernel_name_string>    // [REQUIRED]

 // ------- Register requirements --------
 .reg_count_total    <uint>      // [REQUIRED] a more direct way to specify the parameters
required
 .reg_count_payload  <uint>      // [REQUIRED] rather than indirectly adding the
                          // the payload and temps together to get the total (as is the case
now)
                          // Note: no more "reg-count-temp"

 // -------------- Defaults ---------------
 <default…>               // these should be specified per-kernel and have only kernel-scope
 <default…>               // Same defaults as those already defined in the ISA doc, but just
 <default…>               // moved within the kernel to make each kernel completely self-
sufficient
                          // and not impacted defaults of earlier kernels

 // --------- Memory Requirements ---------
 // [optional] memory block info (just a placeholder for now...)

 <MBDa>                   // memory block descriptor a (TBD)
 <MBDb>                   // memory block descriptor b (TBD)
 <MBDc>                   // memory block descriptor c (TBD)
 <MBDd>                   // memory block descriptor d (TBD)

 // ---------------- Code  ----------------
 .code                           // [REQUIRED]
   <instruction>
   <instruction>
   <instruction>
 <LabelLine>                     // labels are code-block scope
   <instruction>
```

```
  <instruction>
.end_code                  // [REQUIRED]

.end_kernel                        // [REQUIRED]

// --------- next kernel -------------

// --------- next kernel -------------

// ...
```

Usage Examples

This topic is currently under development.

Vector Immediate

The immediate form of vector allows a constant vector to be in-lined in the instruction stream. An immediate vector is denoted by type v as *imm32:v*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. Each 4-bit subfield contains a signed integer value. Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following figure.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
V7	V6	V5	V4	V3	V2	V1	V0

Supporting DirectX 10 Pixel Shader Indexing

When a DirectX 10 Pixel Shader program is converted to run on GEN in channel-serial mode at 16 pixels in parallel, the per-pixel index must be translated into 16 indices with per channel offset. The creation of the per-channel offset can be achieved using the vector immediate.

Consider a generic DirectX 10 Pixel Shader instruction in the form of

opr4r[ind]r2

and assume that r0-r1 contain the 16 indices packed every other words, and r2-r3 contains source 1 and r4-r5 contain the destination. This instruction can be converted into the following GEN instructions. The corresponding operations are illustrated in *Supporting DirectX 10 Pixel Shader Indexing*.

mov (16) r11.0<1>:w 0x01234567:v// assigning a ramp vector, repeated once

mul (16)acc0:wr11.0<0;16,1>:w4:w// expand ramp range to 4 bytes per step

mac (16)r10.0<1>:wr0.0<16;8,2>:w32:w// r10 = index*32 + 0|4|...|28|0|4...|28

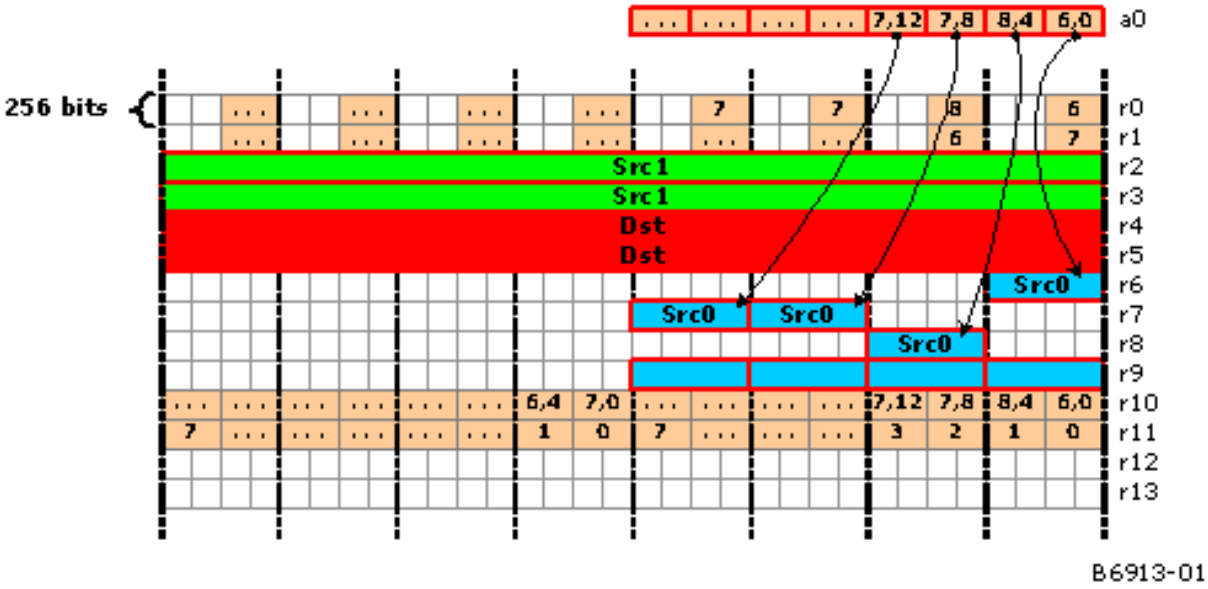
mov (8)a0.0<1>:wr10.0<0;8,1>:w

op (8)r4.0<1>:fr[a0.0]<1,0>:fr2.0<0;8,1>:w// Operate on the first half

mov (8)a0.0<1>:wr10.8<0;8,1>:w// Index values are off by a reg (32b)

op (8)r5.0<1>:fr[a0.0+32]<1,0>:fr3.0<0;8,1>:w// Operate on the second half.

Pixel Shader example using vector immediate.



Without vector immediate support, such translation has to either use a long sequence of scalar instructions which is very inefficient or use a constant load which requires additional constant to be managed in memory.

Supporting OpenGL Vertex Shader Instruction SWZ

When an OpenGL Vertex Shader program is converted to run on GEN in Vertex Pair, i.e. two 4-wide vectors in parallel, the special OpenGL Shader instruction SWZ (Swizzle) needs to be emulated. OpenGL SWZ instruction uses an extended swizzle control field that, in addition to the 4-wide full swizzle control, also includes constant 0 and 1 replacement as well as per channel sign reversal. The later two are not supported by the GEN native instruction. The vector immediate can significantly reduce the overhead of emulating such OpenGL instruction.

Consider an OpenGL Shader instruction in the form of

```
SWZ r1 r0.0-zx-1 // Expected results: r1.x = 0; r1.y = -r0.z; r1.z = r0.x; r1.w = -1
```

It can be emulated by the following three GEN instructions.

```
mul (8) r1.0<1>:f r0.xzxz 0x1F111F11:v // Constant vector of (1 -1 1 1 1 -1 1 1)
mov (1)f0.0 8b'10011001                // Set flag & masked out channels y and z
(f0.0)mov(8) r1.0<1>:f 0x000F000F:v    // Constant vector of (0 0 0 -1 0 0 0 -1)
```

In case that only 0, 1, -1 channel replacement is used and there is no signed swizzle, it may be emulated in two GEN instructions. This is illustrated by the following example:

OpenGL:

```
SWZ r1 r0.0zx-1 // Expected results: r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1
```

GEN:

```
mov (1)f0.0 8b'01100110                     // Set flag and masked out channels x and w
(f0.0)sel (8) r1.0<1>:f r0.yzxy 0x000F000F:v // Constant vector of (0 0 0 -1 0 0 0 -1)
```

Destination Mask for DP4 and Destination Dependency Control

The following example demonstrates the use of destination mask mode of floating point dot-product instruction as well as the use of destination dependency control to improve performance (i.e., avoiding unnecessary thread switch due to possible false dependencies).

Consider a generic DirectX 10 Vertex Shader macro of matrix-vector product that is implemented on GEN in the pair of 4-component vector mode. The DirectX 10 equivalent Shader instructions are as the following.

dp4 r5.x r0 r4

dp4 r5.y r1 r4

dp4 r5.z r2 r4

dp4 r5.w r3 r4

With destination dependency control, the GEN instructions are as the following. The first instruction in the sequence checks for the destination dependency, but does not clear the dependency bit. The subsequent two instructions would do neither of them. The last instruction avoids checking the destination dependency, but at completion, it clears the destination scoreboard. It ensures that the content of the destination register is coherent, if any of the following instructions uses the same register as source.

dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr}

dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}

dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}

dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f {NoDDChk}

Just as a comparison, IF GEN DP4 implies reduction at the destination; additional shifted moves are required to achieve the same results. The corresponding codes are as the following. The lower performance due to the additional three move instruction as well as added back-to-back dependencies shows that why we choose to implement the destination channel replication for floating point DP4.

dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f

mov (1) r5.1<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f

mov (1) r5.2<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f

mov (1) r5.3<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f

Null Register as the Destination

Null register can be used as the destination for most of the instructions. Here are some example usages.

- Null as destination for regular ALU instructions: As all ALU instructions can be configured to update the flag registers using the conditional modifiers, it is not necessary to have a destination register if the programmer only cares about the conditionals of the operation. In that case, a null in the destination operand field saves register space as well as one less dependency checking.
- Null as the destination for SEND/STOR instructions: for the send instruction that only send messages out to an external unit and does not require any return data or feedback, a null in the destination register field signifies the case.
    - One extension of such case is that even though the operation does not have any return values, a return phase with no payload but simply updating the scoreboard flag for a non-null register can provide a signaling mechanism between the thread and the target external unit. One application of this usage is to allow software to manage the coherency of shared memory resources such like the many caches in the system (particularly, valuable for read/write caches).

Use of LINE Instruction

LINE instruction is specifically designed to speed up floating point vector/matrix computation when a program operates in channel serial.

The following example demonstrates how to use LINE instruction to compute Line Equations for DirectX 10 Pixel Shader. In this example, 2 sets of (Cx#, Cy#, Don't Care, C0#) 4-tuple coefficient vectors are stored in registers R1.

R1: Cx0 Cy0 DC Co0 Cx1 Cy1 DC Co1

8 sets of coordinate 2-D vectors (X, Y) are stored in R2 and R3 in the channel serial mode as

R2: X0 X1 … X7

R3: Y0 Y1 … Y7

The objective is to compute the following two line equations for each set of 2D coordinate and store the results in R4 and R5 as

R4: (X0*Cx0 + Y0*Cy0+Co0) … (X7*Cx0 + Y7*Cy0+Co0)

R5: (X0*Cx1 + Y0*Cy1+Co1) … (X7*Cx1 + Y7*Cy1+Co1)

Example LINE Equations

```
//-------------
// Example compute LINE equation in channel serial scenario
//-------------
line (8) acc:f  r1<0;1,0>:f  r2<0;8,1>:f// does acc = X# * Cx0 + Co0
```

mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does r4.# = Y# * Cy0 + acc.#

line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f// does acc = X# * Cx0 + Co0

mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does r4.# = Y# * Cy0 + acc.#

The next example is to compute homogeneous dot product for OpenGL pixel shader running in Channel Serial. In this example, an original OpenGL PS instruction is like

dph R2.x R0 R1

With register remapping, we can store the input coefficient vector R0 in original format in r0, but 8 sets of input coordinate vectors in channel serial format in r2, r3, r4 and r5, and the destination R2.x component in r6.

r0: Cx0 Cy0 Cz0 Co0 DC DC DC DC

r2: X0 X1 … X7

r3: Y0 Y1 … Y7

r4: Z0 Z1 … Z7

r5: W0 W1 … W7

The objective is to compute the following DPH equations and store the results in r6 as

R6: (X0*Cx0+Y0*Cy0+Z0*Cz0+Co0) … (X7*Cx0+Y7*Cy0+Z7*Cz0+Co0)

Example Homogeneous Dot Product in Channel Serial

//-------------

// Example compute homogeneous dot product in channel serial scenario

//-------------

line (8) acc:f r0<0;1,0>:f r2<0;8,1>:f// does acc = X# * Cx0 + Co0

mac (8) acc:f r0.1<0;1,0>:f r3<0;8,1>:f// does acc.# = Y# * Cy0 + acc.#

mac (8) r6<1>:f r0.2<0;1,0>:f r4<0;8,1>:f// does r6.# = Z# * Cz0 + acc.#

Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of DirectX Shader instructions.

Channel Enables for Extended Math Unit

The following example demonstrates how to use the SEND instruction to get service from the Extended Math unit.

Let's consider COS instruction in DirectX 10 in the following form

[([!]p0.{select|any|all})] cos[_sat] dest[.mask], [-]src0[_abs][.swizzle]

For a SIMD4x2 VS implementation with the following register mappings

p0 =>f0.0

src0 =>r0

dest =>r1

The equivalent GEN instruction is as the following

[([!]f0.0.{select|any4h|all4h})] SEND (8) r1[.mask]:f m0 [-][(abs)]r0[.swizzle]:f MATHBOX|COS[|SAT]

If the source swizzle is replication, the message description field can be modified to MATHBOX|COS|SCALAR to take advantage of the fast mode (scalar mode) supported by the Extended Math. The implied move of the SEND instruction is equivalent to the following instruction:

MOV (8) m0[.mask]:f [-][(abs)]r0.0[.swizzle]:f {NoMask}

For a SIMD16 PS implementation, the register mappings are as the followings

p0 =>f0…f3 // in order of R, G, B, A

src0 =>r0,r1; r2,r3; r4,r5; r6,r7

dest =>r8,r9; r10,r11; r12,r13; r14,r15

There are several ways to translate the DirectX instruction, depending on the operand/instruction modifiers present in the DirectX instruction. If predicate is not present and the source swizzle is replication, say, src0.y, which is r2-r3, the translation could be as the following instructions

send (8) r8:f m0 -(abs)r2:f MATHBOX|COS

send (8) r9:f m1 -(abs)r3:f MATHBOX|COS {SecHalf}// use the second half of 8 flag bits

mov (16) r10:fr8:f // All destination color chan's are same

mov (16) r12:fr8:f // MOV is faster than most MathBox func's

mov (16) r14:fr8:f // These MOV's are compressed instructions

Notice that instead of issuing Extended Math messages with the same input data, destination color channel replication is performed by the MOV instructions. This is faster for the thread for most cases as many Extended Math functions consume multiple cycles. This also conserves message bus bandwidth as well as the usage of the shared resource – Extended Math. The destination mask in the DirectX 10 instruction indicates which of the r8 to r15 registers are updated. If the source swizzle is not replication, there will be 8 SEND instructions.

With predication on, if the predication modifier is p0.select, translation is to take the selected flag register f#. The other predication modifiers '.any' and '.all' are translated into '.any4v' and '.all4v', respectively. Notice that with predication on, it is not required to run all 4 pixels in a subspan in the same way, so no need to enforce .any4h/.any4v. The following example shows the instruction with predication (but without .select modifier).

(f0[.any4v|.all4v]) send (8) r8:f m0 -(abs)r2:f MATHBOX|COS

(f0[.any4v|.all4v]) send (8) r9:f m1 -(abs)r3:f MATHBOX|COS {SecHalf}

(f1[.any4v|.all4v]) mov (16) r10:fr8:f // All destination color chan's are same

(f2[.any4v|.all4v]) mov (16) r12:fr8:f // MOV is faster than most MathBox func's

(f3[.any4v|.all4v]) mov (16) r14:fr8:f // These MOV's are compressed instructions

The same instructions works also for predication with select component modifier. We simply replase f0 to f3 above by the selected flag register, say, f1. The modifier of any4h/all4v would also work.

Channel Enables for Scratch Memory

The following example demonstrates how to use the SEND instruction to get service from the Data Port for scratch memory access.

Let's consider general instruction in DirectX 10 that uses scratch memory as a source operand

[([!]p0.{select|any|all})] add dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

For a SIMD4x2 VS implementation with the following register mappings

p0 =>f0

src0 =>r0

src1 =>s2 / r10

dest =>r1

In this example, the scratch memory offset is provided by an immediate and a GRF register r10 is used as the intermediate GRF location for spill/fill of scratch buffer accesses. This arithmetic instruction is converted into a Data Port read followed by an arithmetic instruction.

mov (8) r3:d r0:d {NoMask}// move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}// s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask}// s2 for vertex 1

send (8) r10 m0 r3 DATAPORT|RC|READ_SIMD2

[([!]f0.{sel|any4h|all4h})] add (8) r1[.mask]:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r10[.swizzle]:f

So if scratch register is the source, there is no need to use the channel enable side band. This is also true for channel-serial PS cases.

Now, let's consider the case when a scratch register is the destination of an instruction.

p0 =>f0

src0 =>r0

src1 =>r1

dest =>s2 / r10

We have

add (8) m1:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r1[.swizzle]:f

mov (8) r3:d r0:d {NoMask}// move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}// s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask}// s2 for vertex 1

```
[([!]f0.{sel|any4h|all4h})] send (8) null[.mask] m0 r3 DATAPORT|RC|WRITE_SIMD2
```

Notice that with a null as the posted destination register, we are able to transfer the [.mask] over the message channel enables. In many cases for scratch memory assess, a write-with-commit is required, therefore, the posted destination register could be r10.

Now, let's consider the PS case when a scratch register is the destination of an instruction.

p0 =>f0-f4

src0 =>r0-r7

src1 =>r8-r15

dest =>s16-s23 / r16-r23

When predication is not on (or predication with swizzle control on), we have

add (16) m4:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m6:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m8:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m10:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (8) r3:d 0x76543210:v {NoMask}// ramp function

mul (16) acc0:d r3:d 16 {NoMask}// ramp function

add (8) acc0:d acc0:d 64 {NoMask,SecHalf}// ramp function

add (16) m2:d acc0:d 2*256 {NoMask}// ramp function

```
send (16) null m1 r3 DATAPORT|RC|WRITE_SIMD16
```

As there is no bit left from the unit specified descriptor field, the 4 bit mask must be put into the header field in m1, which requires at least two more instructions.

Alternatively, or for the case that predication without modifier is on, we can do a read-modify-write.

mov (8) r3:d 0x76543210:v {NoMask}// ramp function

mul (16) acc0:d r3:d 16 {NoMask}// ramp function

add (8) acc0:d acc0:d 64 {NoMask,SecHalf}// ramp function

add (16) m2:d acc0:d 2*256 {NoMask}// ramp function

```
send (16) r16 m1 r3 DATAPORT|RC|READ_SIMD16 // read from scratch
```

// some of the following four instructions may be omitted based on [.mask] field

[([!]f0.{sel|any4v|all4v})] add (16) r16:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r18:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r20:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r22:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (16) m4:f r16:f {NoMask}

mov (16) m6:f r18:f {NoMask}

mov (16) m8:f r20:f {NoMask}

mov (16) m10:f r22:f {NoMask}

```
send (16) null m1 null DATAPORT|RC|WRITE_SIMD16 {NoMask}// write back to scratch
```

Flow Control Instructions

Unconditional branches are performed through direct manipulation of the 32-bit IP architectural register. For example:

```
mov (1) IP <memory_address>// jump absoluteadd (1) IP IP <byte_count>// jump relative
```

Note that jump distances are specified in terms of bytes, as opposed to instruction counts in the case of *break*, *halt*, etc. To minimize confusion, an assembler-only instruction 'jmp <inst_count>', where <inst_count> is an immediate term, may be defined which takes an instruction count for a distance. The *jmp* pseudo-opcode can be mapped to an "add (1) ip ip <inst_count> * 16" instruction.

IP is aligned to an 8-byte boundary, thus the 3 LSBs are not maintained in the IP architectural register and should not be relied upon by software.

IP, when used as a source operand, reflects the memory address of the instruction in which it is used. The following are examples illustrating the use of IP:

```
add (1) IP4*16// jumps to HERE_1
add (1) IP0x35// jumps to HERE_1 (4 lsb's don't-care)  <instruction>
 <instruction>

HERE_1:<instruction>HERE_2:<instruction>
 <instruction>
 add (1) IP -2*16// jumps to HERE_2 ...
 add (1) IP 0// infinite loopadd (1) IP 0xF// infinite loop ...
```

Note for Assembler: The if/iff/else/while/break instructions identify relative addresses as the targets of an implicit jump associated with the instruction. These are optional in the assembly syntax as the jitter can determine the location of the matching instruction (e.g. matching endif instruction for a given if instruction).

Execution Masking

This topic is currently under development.

Branching

Example. If / Else / EndIf

```
//-------------
// Example if/else/endif scenario
// "if (r5==r4) ...else ... end-if"
//-------------
...
cmp.e.f0 (8) null r5 r4// does r5 == r4?
(f0) if (8)  HERE_1// "if" part - save then update IMASK;
//   or goto the 'else' if all false
...
...
HERE_1:// now do the 'else' part
else (8) HERE_2// "else" part - invert IMASK
//   or goto the 'endif' if all false
...
...
HERE_2:
endif// "end-if" part – restore IMASK
....// and continue...
```

If it is known that the code has no nested conditionals, a predicate can be used for a lower overhead, more efficient if/else/endif. (One must consider the probability of all channels taking the same branch, and the number of instructions under the if/else blocks as to which conditional method, predicate or mask, is most efficient).

Fast-If

Below is an example of a fast-if instruction. For the 'iff' instruction, only and iff-endif construct is allowed, as opposed to a if-else-endif. Note that the target address for branching if all enabled channels fail is one instruction beyond the endif, as the 'iff' does not push and update the IMask unless the branch is taken for at least one execution channel.

Example Fast If

```
//-------------
// Example – Fast If
//One instruction overhead conditional
//-------------
...
cmp.e.f0 (8) null r5 r4// any flag update
...
(f0)iff (8) HERE_1// "fast-if" – only pushes IMask;
//   if execution falls through,
//   else go to HERE_1
...
...
endif// "end-if" part – restores IMask
HERE_1:
...// and continue...
```

Cascade Branching

As there is no 'elseif' instruction, a C-like cascade branching such as if / elseif / else / endif, can be realized using the basic building blocks of if / else / endif as shown in the following example. Notice that two 'endif's' are required in order to pop the IStack correctly.

Example. If / Elseif / Else / EndIf

```
//-------------
// Example if/elseif/else/endif scenario
// "if (r5==r4) ...elseif (r6>r7) else ... end-if"
//-------------
...
cmp.e.f0 (8) null r5 r4// does r5 == r4?
(f0)if (8) HERE_1// "if" part - save then update IMask;
//   or go to the 'else' part if all false
...
...
```

```
HERE_1:// now do the ‘else’ part
else (8) HERE_2// “else if” part - invert IMask
//   or go to the ‘else’ part if all false
cmp.g.f0 (8) null r6 r7// is r6 > r7?
(f0)if (8) HERE_3// “if” part - save then update IMask;
//   or go to the ‘else’ part if all false
...
...
HERE_3:// now do the ‘else’ part
else (8) HERE_4// “else” part - invert IMask
//   or go to the ‘end-if’ part if all false
...
...
HERE_4:
endif// “end-if” part – restore IMask for elseif
HERE_2:
endif// “end-if” part – restore IMask for if
....
```

Compound Branches

Compound branches are supported through the ability logically combine flag registers for each intermediate result.

Example Compound Branch

```
//-------------
// Example: “if (r0 > r1) OR (r2 <= r3)”
//-------------
...
cmp.g.f0 (8) null r0:d r1:d// r0 > r1?
cmp.le.f1 (8) null r2:d r3:d// r2 <= r3?
or (1) f0:w f0:w f1:w// combine f0 and f1
(f0) if (8) HERE_1// Can now do normal if/else
```

...

...

HERE_1:endif

...

Example Compound Branch Using 'Any' or 'All'

```
//-------------
// Example: assuming we're doing a channel-serial vector in r0-r3
// We want to know if all components of the vector are > 0x80
//-------------
...
cmp.g.f0 (16)  null r0 0x80// r0 > 0x80?
cmp.g.f1 (16)  null r1 0x80// r1 > 0x80?
cmp.g.f2 (16)  null r2 0x80// r0 > 0x80?
cmp.g.f3 (16)  null r3 0x80// r1 > 0x80?
(f0.all4v) if (16) HERE_1
...
...// code executed only for those channels
...// where per-channel r0,r1,r2,r3 all > 0x80
...
HERE_1:endif
...// and continue...
```

Looping

Due to GEN's SIMD-16 architecture, it must support the case of up to 16 loops running in parallel. These must be handled as independent loops, each with its own loop-exit condition which could occur after a different number of loop iterations. To account for each channel's progress, a 16b loop-mask 'LMask' is defined with 1b associated to each execution channel. This mask keeps track of which channels remain active inside a loop block.

Basic Do-While Loop

Looping illustrates the most basic loop. Two operations must be accomplished before loop entry. (1) Prior to loop entry, there is some subset of enabled channels as dictated by the code sequence prior. In general, the active status of each channel is indicated in the virtual EMask any point in time. These active channels will become the channels over which the loop is run, and LMask must be initialized with the

EMask value. (2) Since a given loop may be nested within another loop, the previous LMask & CMask must be saved to the LStack for later restoration upon loop completion. The 'msave' instruction performs both the save and update in a single instruction, and thus all loop-blocks should be fronted with a "msave LStack LMask" and "msave LStack CMask" operation.

Note that the LMask and CMask share the same mask-stack. Thus, CMask must always be a 1's-subset of the LMask for proper stack operation. This is the case if CMask is updated to LMask each pass through the loop (see *Looping*) and through the 'break' instruction updating both masks.

Each pass through the loop, a loop terminating operation must be evaluated and stored in a flag register. This condition must be evaluated on a channel-by-channel basis as exemplified:

cmp.z.f0(8) null r2 d3// any operation that updates a flag

The result of this operation sets a bit per channel in the specified flag register, which is then used in the 'while' instruction. As loops are performed, channels may become disabled as their termination condition is met.

'While' termination is determined on a channel-by-channel basis by the logical AND of corresponding bit positions of AMask, CMask and the specified flag. If the result is '1' the channel remains enabled for the next pass of the loop; if '0' the channel is disabled until loop fall-through. The 'while' instruction causes the LMask to be updated with the latest result of enabled channels. If any channel remains enabled (LMask != ...000b), an additional pass through the loop is made. Once a channel is terminated for the loop operation, it remains terminated until the loop is complete for all channels.

Upon fall through, the 'while' instruction causes the previously saved LMask & CMask to be popped from the LStack, enabling execution on the same subset of channels enabled prior to loop entry (unless a channel had been otherwise terminate inside the loop via 'halt').

Example Basic Loop Construct

```
//-----------------------
//Example: Basic do-while loop structure
//-----------------------
...
do// save L/CMask & update
BEGIN_LOOP:
mov (1) CMask LMask{NoMask}// update CMask for this pass
...
...
<some flag update>
()while (8) BEGIN_LOOP// cond. branch
//   + restores LMask on fall-through
```

...

Do-While Loop with Break

A loop may also be terminated for any channel via the 'break' instruction. The 'break' instruction causes the corresponding bit positions of enabled channels to be cleared in the LMask. If the updated LMask = ...000b, a branch is made to the specified instruction location. An example is shown below in which the 'break' is at the same conditional-nesting level as the terminating 'while'. Its primary value may simply be to support a "do...break.. while (true)" –type structure for a more direct 1:1 translation from higher-level source code.

Example Loop Construct With Non-Nested 'Break'

```
//-------
//Example: While-true loop
//-------
#define BrkCode(i,d)(i « 16) + d
do// save L/CMask & update
BEGIN_LOOP:
mov (1) CMask LMask{NoMask} // update CMask for this pass
...
<some flag update>
()break (8) BrkCode(0,HERE_1)// Restores LMask when all
// channels complete loop.
...
...
while (8) BEGIN_LOOP// while true
HERE_1:
...
```

A break condition may occur from various levels of nested-ifs. This gives rise to the possibility that a the loop may terminate from within nested 'if's, and due to the jump inherent in the 'break' instruction, the associated 'endif's are not encountered to clean-up the IStack as nesting levels are exited.

Example Loop Construct With 'Break' From Within Nested If's

```
//-------
//Example: General Loop Structure w/ break inside if's
//-------
```

```
#define BrkCode(i,d)(i « 16) + d
do// save L/CMask & update
BEGIN_LOOP:
mov (1) CMask LMask{NoMask} // update CMask for this pass
...
if ...
if ...
if ...
...
()break (8) BrkCode(3,HERE_1)// we're 3 levels deep, so
...
endif
endif
endif
...
()break (8) BrkCode(0,HERE_1)
...
while (8) <flag_spec> BEGIN_LOOP// cond. branch
// + restores C/LMask on fall-through
HERE_1:
```

Do-While Loop with Continue

A continue instruction 'cont' is provided skip to the next iteration of the loop. Because not all channels participating in the loop may be enabled at the time this instruction is executed, some channels may require continuation of the loop. A special mask 'CMask' is defined which accounts for channels temporarily disabled for the current loop pass.

Since loops may nested, the CMask must be saved and restored around a loop similar to LMask. Since the CMask value within a properly constructed loop is always a subset of the LMask, it can share the LStack for storage, so long as it is pushed after LMask as shown in *Looping*. This save/restore operations are not required if the loop being entered does not have any occurrence of a continue instruction.

Example Do-While with Continue

```
//-------
//Example: General Loop Structure w/ basic break and cont.
```

```
//-------
#define ContCode(i,d)(i « 16) + d
do// save L/CMask & update
BEGIN_LOOP:
mov (1) CMask EMask// re-initialize CMask for this pass
...
...
() cont (8) ContCode(0,HERE_1)
...
HERE_1:
()while (8) BEGIN_LOOP// cond. branch
//   + restores C/LMask on fall-through
...
```

Indexed Jump

Example Indexed Jump

```
//-------------
// Code example shows the use of jmpi to perform a case statement
// of any number of options in 3 jumps
//-------------
.default_execution_size 8
...
jmpi r0<0,1,0>// jump relative, based on r0.a.x
// ----- Jump Table ------
jmp HERE_0// redirect for case 0
jmp HERE_1// redirect for case 1
jmp HERE_2// redirect for case 2
jmp HERE_3// redirect for case 3
...
HERE_0:// ... case 0 ...
```

...

jmp DONE

HERE_1:// ... case 1 ...

...

jmp DONE

HERE_2:// ... case 2 ...

...

jmp DONE

HERE_3:// ... case 3 ...

...

DONE:

...// and continue...