

Intel® Open Source HD Graphics and Intel Iris™ Graphics

Programmer's Reference Manual

For the 2014-2015 Intel Core™ Processors, Celeron™ Processors
and Pentium™ Processors based on the "Broadwell" Platform

Volume 7: 3D Media GPGPU

May 2015, 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

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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

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

Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2015, Intel Corporation. All rights reserved.

Table of Contents

Registers in Render Engine	1
Predication	1
Predicate Render Registers.....	1
MI_SET_PREDICATE	2
MI_PREDICATE	2
Predicated Rendering Support in HW	2
Registers in Render Engine	4
Context Save Registers.....	5
VF Instance Count Registers.....	5
Mode and Misc Ctrl Registers	6
L3CNTLREG - L3 Control Register	6
Pipelines Statistics Counter Registers.....	7
CS_CTX_TIMESTAMP- CS Context Timestamp Count:.....	8
AUTO_DRAW Registers	9
MMIO Registers for GPGPU Indirect Dispatch.....	10
CS ALU Programming and Design	11
Generic Purpose Registers.....	11
ALU BLOCK Diagram.....	12
Instruction Set.....	13
Instruction Format	13
LOAD Operation	13
Arithmetic/Logical Operations.....	14
STORE Operation.....	14
Summary for ALU	15
Summary of Instructions Supported	15
Table for ALU OPCODE Encodings.....	15
Table for Register Encodings	17
CS_GPR - Command Streamer General Purpose Registers.....	18
Memory Interface Commands for Rendering Engine.....	18
State Commands	18
STATE_BASE_ADDRESS	18
Synchronization of the 3D Pipeline.....	19

Top-of-Pipe Synchronization	19
End-of-Pipe Synchronization	19
Synchronization Actions.....	20
Writing a Value to Memory.....	20
PS_DEPTH_COUNT	20
Generating an Interrupt.....	21
Invalidating of Caches	21
PIPE_CONTROL Command.....	22
PIPE_CONTROL.....	23
Programming Restrictions for PIPE_CONTROL	23
Post-Sync Operation	24
Flush Types.....	25
Stall.....	25
Render Logical Context Data.....	26
Overall Context Layout.....	26
Context Layout.....	26
Register State Context.....	27
Register State Context	28
Ring Buffer.....	40
Ring Context.....	40
The Per-Process Hardware Status Page	41
Render Engine Power Context	43
Command Ordering Rules.....	45
PIPELINE_SELECT.....	46
PIPE_CONTROL.....	46
Common Pipeline State-Setting Commands.....	46
3D Pipeline-Specific State-Setting Commands	47
Media Pipeline-Specific State-Setting Commands	47
3DPRIMITIVE.....	47
MEDIA_OBJECT	47
Resource Streamer.....	48
Resource Streamer Sync Commands	48
Introduction.....	48

Common Abbreviations	49
Theory of Operation.....	49
Resource Streamer Functions.....	49
Detailed Resource Streamer Operations	51
Introduction.....	51
Resource Streamer Operation Descriptions	51
Batch Processing	51
Context Save	51
HW Binding Table Image	52
Gather Push Constants Image.....	52
Push Constant Image.....	53
HW Binding Table Generation.....	54
Gather Push Constants	55
Constant Buffer Generation (not DX9)	56
Commands Actions in the RS.....	57
Resource Streamer Programming Guidelines.....	64
RS Interactions with the 3D Command Streamer.....	64
RS Interactions with Memory Requests.....	64
Fundamental Programming and Operational Assumptions	65
Non-Operational Activities.....	65
Hardware Binding Tables	65
3DSTATE_BINDING_TABLE_POOL_ALLOC.....	66
Gather Constants.....	67
Dx9 Constant Buffer Generation	68
Vertex Shader Constant	69
Pixel Shader Constant.....	70
Shared Functions	71
3D Sampler	71
Sampling Engine	72
Texture Coordinate Processing	74
Texture Coordinate Normalization	74
Texture Coordinate Computation	75
Texel Address Generation	76

Level of Detail Computation (Mipmapping)	76
Base Level Of Detail (LOD)	77
LOD Bias	77
LOD Pre-Clamping	78
Min/Mag Determination	78
LOD Computation Pseudocode	79
Intra-Level Filtering Setup	80
MAPFILTER_NEAREST	80
MAPFILTER_LINEAR	81
MAPFILTER_ANISOTROPIC	82
MAPFILTER_MONO	83
Inter-Level Filtering Setup	84
Texture Address Control	85
TEXCOORDMODE_MIRROR Mode	86
TEXCOORDMODE_MIRROR_ONCE Mode	87
TEXCOORDMODE_WRAP Mode	88
TEXCOORDMODE_CLAMP Mode	89
TEXCOORDMODE_CLAMPBORDER Mode	90
TEXCOORDMODE_CUBE Mode	91
Texel Fetch	91
Texel Chroma Keying	91
Chroma Key Testing	91
Chroma Key Effects	92
Shadow Prefilter Compare	92
Texel Filtering	93
Texel Color Gamma Linearization	93
Multisampled Surface Behavior	94
Multisample Control Surface	94
State	95
Surface State Fetch	95
Sampler State Fetch	95
State Caching	96
SURFACE_STATE	96

Sampler Output Channel Mapping	97
SURFACE_STATE for Deinterlace sample_8x8 and VME	101
SAMPLER_STATE	102
Border Color Programming for Integer Surface Formats	104
Messages	105
Message Descriptor and Execution Mask	105
Execution Mask	105
Message Descriptor	106
Message Header	106
3D Sampler Message Types	109
Common Message Variants	110
Restrictions and Programming Notes for All Message Types	110
Sample Message Types	111
Id Message Types	114
gather4 Message Types	116
gather4 Message Type Description	116
Supported Variants:	116
Restrictions and Programming Notes for gather4:	117
Restrictions and Programming Notes for gather4_c:	117
Restrictions and Programming Notes for gather4_po:	118
Restrictions and Programming Notes for gather4_po_c:	118
sampleinfo Message Type	119
sampleinfo Message Definition	119
Supported Variants:	119
LOD Message Type	119
LOD Message Definition	119
Supported Variants:	119
Restrictions and Programming Notes for LOD:	119
resinfo Message Type	120
resinfo Message Definition	120
Supported Variants:	120
Restrictions and Programming Notes for resinfo:	121
cache_flush Message Type	121

cache_flush Message Definition	121
Supported Variants:.....	121
Restrictions and Programming Notes for cache_flush:.....	121
Media Message Types	122
sample_unorm Message Types	122
sample_unorm Message Definition.....	122
Supported Variants:.....	122
Restrictions and Programming Notes for sample_unorm, sample_unorm_RG, sample_unorm_killpix, sample_unorm_RG_killpix:.....	123
sample_8x8 Message Type	124
Supported Variants:.....	124
Restrictions and Programming Notes for sample_8x8:	124
Message Format.....	124
Supported SIMD Types.....	124
Message Length.....	125
Response Length.....	125
Message Formats.....	127
Parameter Types.....	129
SIMD Payloads	130
Writeback Message	132
SIMD16.....	132
Return Format = 32-bit.....	132
Return Format = 16-bit.....	133
SIMD8/SIMD8D	134
Return Format = 32-bit.....	134
Return Format = 16-bit.....	135
SIMD4x2	136
Shared Functions – Data Port.....	137
Data Cache	138
Sampler Cache.....	139
Surfaces	139
Surface State Model.....	139
Stateless Model	139
Shared Local Memory (SLM)	140

Write Commit.....	141
ReadWrite Ordering	141
Accessing Buffers.....	142
Accessing Media Surfaces.....	143
Boundary Behavior	143
State.....	144
BINDING_TABLE_STATE	144
SURFACE_STATE	144
Messages	144
Global Definitions	144
Data Port Messages	145
Message Descriptor.....	147
Message Descriptor	147
Message Header	152
Write Commit Writeback Message	153
OWord Block ReadWrite	154
Message Descriptor.....	155
Message Payload (Write).....	155
Writeback Message (Read)	156
Unaligned OWord Block Read	156
Message Descriptor.....	157
Writeback Message (Read)	157
OWord Dual Block ReadWrite	158
Message Descriptor.....	159
Message Payload.....	159
Additional Message Payload (Write)	159
Writeback Message (Read)	160
Media Block ReadWrite	160
Message Descriptor.....	163
Message Header	164
Message Payload (Write).....	168
Writeback Message (Read)	168
DWord Scattered ReadWrite.....	169

Message Descriptor.....	170
Message Payload.....	170
Additional Message Payload (Write)	171
Writeback Message (Read).....	172
Message Descriptor.....	173
Message Payload.....	174
Source Payload.....	174
Writeback Message	175
Byte Scattered ReadWrite.....	176
Message Descriptor.....	177
Message Payload.....	177
Additional Message Payload (Write)	178
Writeback Message (Read).....	179
TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation.....	180
Typed Surface ReadWrite Message Descriptor	186
Typed Surface Read/Write Message Descriptor	187
Untyped Surface ReadWrite Message Descriptor	188
Typed Atomic Operation Message Descriptor	189
Typed Atomic Operation SIMD4x2 Message Descriptor.....	190
Untyped Atomic Operation Message Descriptor	191
Untyped Atomic Operation SIMD4x2 Message Descriptor.....	192
Atomic Counter Operation Message Descriptor	193
Atomic Counter Operation SIMD4x2 Message Descriptor.....	194
Message Header	195
Message Payload.....	196
SIMD16 Address Payload	197
SIMD16 Source Payload (Atomic Operation Message Only)	198
SIMD16 Source Payload (AOP_CMPWR8B Only)	199
SIMD16 Write Data Payload (Write Message Only).....	200
SIMD8 Address Payload.....	201
SIMD8 Source Payload (Atomic Operation Message Only).....	202
SIMD8 Write Data Payload (Write Message Only)	203
SIMD8 Write Data Payload (Tile W Write Message Only)	204

SIMD4x2 Address Payload	205
SIMD4x2 Source Payload (Atomic Operation Message Only)	205
SIMD4x2 Source Payload (AOP_CMPWR8B Only)	206
SIMD4x2 Write Data Payload (Write Message Only)	206
Writeback Message	207
SIMD8 DWORD Read	207
SIMD8 QWORD Read	207
SIMD16 Read	208
SIMD8 Read	209
SIMD8 Read (Tile W)	210
SIMD4x2 Read	211
SIMD16 Atomic Operation	212
SIMD16 Atomic Operation (AOP_CMPWR8B Only)	213
SIMD8 Atomic Operation	214
SIMD8 Atomic Operation (AOP_CMPWR8B Only)	214
SIMD4x2 Atomic Operation	215
SIMD4x2 Atomic Operation (AOP_CMPWR8B Only)	215
A64 Scattered Read or Write	216
Message Descriptor	217
Message Address Payload	217
Additional Message Payload (QWord Write)	218
Additional Message Payload (DWord Write)	219
Additional Message Payload (8 Byte Write)	220
Additional Message Payload (1 or 2 or 4 Byte Write)	221
Writeback Message (QWord Read)	222
Writeback Message (DWord Read)	223
Writeback Message (8 Byte Read)	224
Writeback Message (1 or 2 or 4 Byte Read)	225
A64 Untyped Atomic Operation	226
A64 Untyped Atomic Operation Message Descriptor	229
Message Payload	230
SIMD8 Address Payload	230
SIMD4x2 Address Payload	231

SIMD8 Source Payload (QWORD Atomic Operation Message Only)	231
SIMD8 Source Payload (DWORD Atomic Operation Message Only)	232
SIMD8 Source Payload (AOP_CMPWR16B Only)	232
SIMD8 Source Payload (AOP_CMPWR8B Only).....	233
SIMD4x2 Source Payload (QWORD Atomic Operation Message Only)	233
SIMD4x2 Source Payload (DWORD Atomic Operation Message Only).....	234
SIMD4x2 Source Payload (AOP_CMPWR16B Only)	234
SIMD4x2 Source Payload (AOP_CMPWR8B Only).....	235
Writeback Message	235
SIMD8 Atomic Operation (QWord)	235
SIMD8 Atomic Operation (DWord)	236
SIMD8 Atomic Operation (AOP_CMPWR16B Only).....	236
SIMD8 Atomic Operation (AOP_CMPWR8B Only)	237
SIMD4x2 Atomic Operation (QWord).....	237
SIMD4x2 Atomic Operation (DWord).....	238
SIMD4x2 Atomic Operation (AOP_CMPWR16B Only)	238
SIMD4x2 Atomic Operation (AOP_CMPWR8B Only).....	238
A64 Untyped Atomic Float Add Operation	239
Untyped Atomic Float Add Operation Message Descriptor	239
Message Payload.....	240
SIMD8 Address Payload.....	240
SIMD8 Source Payload (QWORD Atomic Operation Message Only)	241
SIMD8 Source Payload (DWORD Atomic Operation Message Only)	241
Writeback Message	242
A64 Untyped Surface Read or Write Operation	242
Untyped Surface Read or Write Message Descriptor	243
Message Payload.....	243
SIMD8 Address Payload.....	244
SIMD4x2 Address Payload	244
SIMD8 DWORD Write Data Payload (Write Message Only).....	245
SIMD4x2 DWORD Write Data Payload (Write Message Only)	245
Writeback Message	246
SIMD8 DWORD Read	246

SIMD4x2 DWORD Read	247
A64 Block Read or Write Operation	247
Message Descriptor.....	249
Message Header.....	250
Message Payload (OWord Write)	250
Writeback Message (OWord Read).....	251
Writeback Message (Unaligned OWord Read).....	251
Message Payload (Dual OWord Write).....	252
Writeback Message (Dual Oword Read).....	252
Message Payload (HWord Write).....	252
Writeback Message (HWord Read).....	253
Untyped Atomic Float Add Operation	253
Untyped Atomic Float Add Operation Message Descriptor	254
Message Header.....	255
Message Payload.....	256
SIMD16 Address Payload	256
SIMD8 Address Payload.....	257
SIMD16/SIMD8 DWORD Source Payload	258
SIMD16/SIMD8 QWORD Source Payload.....	259
Writeback Message	260
SIMD16 SIMD8 DWORD Atomic Float Add	260
SIMD16/SIMD8 QWORD Atomic Float Add.....	261
Read Surface Info.....	261
Message Descriptor.....	261
Address Payload	261
Writeback Message	262
Transpose Read	263
Message Descriptor.....	263
Message Header	264
Writeback Message	265
Scratch Block Read or Write	265
Message Descriptor.....	266
Message Header	267

Message Payload (Write).....	268
Message Payload (Read)	268
Writeback Message (Read).....	268
Memory Fence	268
Message Descriptor.....	269
Message Header	271
Writeback Message	271
Pixel Data Port.....	272
Cache Agents	272
Accessing Render Targets	272
Message Sequencing Summary	272
Single Source	273
Dual Source	274
Replicate Data.....	274
Multiple Render Targets (MRT).....	274
Render Target Read and Write	275
SubspanPixel to Slot Mapping	279
Message Descriptor	281
Message Header	281
Message Header.....	281
Writeback Message (Read)	282
Header for SIMD8_IMAGE_WRITE.....	284
Source 0 Alpha Payload	286
oMask Payload	287
Color Payload SIMD16 Single Source.....	288
Color Payload.....	288
Color Payload SIMD8 Single Source	289
Color Payload SIMD16 Replicated Data	289
Color Payload SIMD8 Dual Source	290
Render Target Read and Write	291
Message Header	294
Shared Functions Pixel Interpolator	295
Messages	295

Initiating Message	295
Message Descriptor.....	295
"Per Message Offset" Message Descriptor.....	296
"Sample Position Offset" Message Descriptor	296
"Centroid Position" and "Per Slot Offset" Message Descriptor.....	297
Message Payload for Most Messages.....	297
SIMD8 Per Slot Offset Message Payload	297
SIMD16 Per Slot Offset Message Payload.....	298
Writeback Message.....	300
SIMD8	300
SIMD16.....	301
Shared Functions - Unified Return Buffer (URB).....	302
URB Size	302
URB Access.....	302
State.....	303
FF_SYNC Messages.....	303
URB Messages	304
Execution Mask.....	305
Message Descriptor	305
URB_WRITE and URB_READ	307
Message Header.....	307
URB_WRITE_HWORD Write Data Payload.....	309
URB_NOSWIZZLE	310
URB_INTERLEAVED.....	311
URB_READ_HWORD Writeback Message	312
URB_NOSWIZZLE	313
URB_INTERLEAVED.....	314
URB_WRITE_OWORD Write Data Payload.....	315
URB_NOSWIZZLE	315
URB_INTERLEAVED.....	316
URB_READ_OWORD Writeback Message.....	317
URB_NOSWIZZLE	318
URB_INTERLEAVED.....	319

URB_ATOMIC	320
Message Header	320
Writeback Message	320
URB_SIMD8_Write and URB_SIMD8_Read	321
Message Descriptor	321
Message Header	322
Per Slot Offset Message Phase	323
Channel Mask Message Phase	324
Write Data Payload	325
Writeback Message	326
Message Gateway	327
Messages	328
Message Descriptor	329
OpenGateway Message	330
Message Payload	331
Writeback Message to Requester Thread	332
CloseGateway Message	333
Message Payload	333
Writeback Message to Requester Thread	333
ForwardMsg Message	333
Message Payload	334
Writeback Message to Requester Thread	336
Writeback Message to Recipient Thread	336
GetTimeStamp Message	337
Message Payload	337
Writeback Message to Requester Thread	338
BarrierMsg Message	339
Message Payload	340
Writeback Message to Requester Thread	341
Broadcast Writeback Message	341
Media Sampler	342
Shared Functions – Video Motion Estimation	342
Theory of Operation	342

Shape Decision	342
Minor Shape Decision Prior to FME	344
Major Shape Decision Prior to FME	346
Shape Update after FME	346
Final Code Decision after BME	346
Early Decisions	347
Surfaces	349
State	349
BINDING_TABLE_STATE	349
SURFACE_STATE	349
VME_STATE	349
VME_SEARCH_PATH_LUT_STATE	350
Software Interface – PRM Highlights	354
Message Structure Overview	354
IME and IDM Message Descriptor	355
Input GRFs	356
Input Message Phases by Type	356
Output GRFs	357
Output Message Phases by Type	358
Binding Table Pointers	359
Progressive Content	359
Interlaced Content	360
Glossary of Messages	361
Universal Input Message Phases	361
SIC Input Message Phases	379
IME Input Message Phases	384
FBR Input Message Phases	388
IDM Input Message Phases	389
Return Data Message Phases	390
IME StreamOut	399
IDM Stream-Out	399
IDM16x16 Streamout Message Format	400
IDM8x8 Streamout Message Format	401

Sample_8x8 State.....	403
SURFACE_STATE for Deinterlace, sample_8x8, and VME	403
SAMPLER_STATE for Sample_8x8 Message.....	403
Media Object Dispatch Pseudocode.....	404
Calculate Residual Blocks Pseudocode	404
Dispatch Media Object Pseudocode.....	406
SIMD32/64 Messages	407
Initiating Message	407
SIMD32 Payload	407
SIMD64 Payload.....	408
Vertical Block Number Restrictions	409
Payload Parameter Definition	410
SIMD32_64 Message Descriptor	412
SIMD32_64 Message Header.....	412
Message Header	412
SIMD32_64 Payload Parameter Definition	414
SIMD32_64 Message Types.....	414
Writeback Message.....	415
SIMD32	415
Sample_unorm*	415
"16 bit Full" Output Format Control Mode	416
"16 Bit Chrominance Downsampled" Output Format Control Mode	417
"8 Bit Full" Output Format Control Mode	419
"8 Bit Chrominance Downsampled" Output Format Control Mode.....	420
Additional Writeback Phase for sample_unorm+killpix	421
Cache_flush	421
Sample_8x8 Writeback Messages	422
"16 bit Full" Output Format Control Mode	423
"16 Bit Chrominance Downsampled" Output Format Control Mode	424
"8 Bit Full" Output Format Control Mode	426
"8 Bit Chrominance Downsampled" Output Format Control Mode.....	427
Sampler_8x8 – Writeback Message for Convolve and 1 Pixel Convolution and 1D Vertical Convolve.....	428
Sampler_8x8 – Writeback Message for MinMaxFilter	429

"16 Bit Full" Output Format Control Mode	429
"8 Bit Full" Output Format Control Mode:	431
Sampler_8x8 – Writeback Message for MinMax	432
Sampler_8x8 – Writeback Message for Dilate or Erode	432
Sampler_8x8 – Writeback Message for Centroid	433
Sampler_8x8 – Writeback Message for BoolCentroid/BoolSum	433
Sampler_8x8 – Writeback Message for AVS	434
"16 bit Full" Output Format Control Mode	435
"16 Bit Chrominance Downsampled" Output Format Control Mode	437
"8 Bit Full" Output Format Control Mode	440
"8 Bit Chrominance Downsampled" Output Format Control Mode	441
SIMD32 Surface State	443
SIMD32 Sampler State	443
3D Pipeline Stages	444
3D Pipeline-Level State	444
Statistics	446
Statistics Gathering	446
3D Pipeline Geometry	448
Block Diagram	448
3D Primitives Overview	449
Vertex Data Overview	457
Vertex URB Entry (VUE) Formats	458
Vertex Positions	461
Clip Space Position	462
NDC Space Position	462
Screen-Space Position	462
Vertex Fetch (VF) Stage	463
State	463
Control State	463
Index Buffer (IB) State	463
Vertex Buffers (VB) State	464
VERTEXDATA Buffers – SEQUENTIAL Access	465
VERTEXDATA Buffers – RANDOM Access	466

INSTANCEDATA Buffers	467
Vertex Definition State	468
Input Vertex Definition	468
3D Primitive Command	469
Functions	470
Input Assembly	470
Vertex Assembly	470
Vertex Cache	471
Input Data: Push Model vs. Pull Model	472
Generated IDs	472
3D Primitive Processing	473
Index Buffer Access	473
Vertex Element Data Path	474
FormatConversion	475
DestinationFormatSelection	478
Dangling Vertex Removal	478
Vertices Generated	479
Objects Generated	479
Vertex Shader (VS) Stage	480
State	480
Functions	480
Vertex Shader Cache (VS\$)	480
SIMD4x2 VS Thread Request Generation	482
SIMD4x2 VS Thread Execution	482
VS Thread Dispatch Masks	483
Vertex Output	483
Thread Termination	483
Primitive Output	483
Statistics Gathering	484
Payloads	485
SIMD4x2 Payload	485
SIMD8 Payload	487
Hull Shader (HS) Stage	489

State	489
Functions	490
Patch Object Staging	490
HS Thread Execution	490
HS Thread Dispatch Mask	490
Patch URB Entry (Patch Record) Output	490
Patch Header DW0-7	490
DOMAIN_POINT Structure	495
Statistics Gathering	495
HS Invocations	495
Payloads	496
SINGLE_PATCH Payload	496
Tessellation Engine (TE) Stage	500
State	500
Functions	501
Patch Culling	501
Tessellation Factor Limits	501
Partitioning	501
Domain Types and Output Topologies	502
QUAD Domain Tessellation	502
TRI Domain Tessellation	504
ISOLINE Domain Tessellation	505
Domain Shader (DS) Stage	506
State	506
Functions	507
SIMD4x2 Thread Execution	507
Statistics Gathering	507
Payloads	508
SIMD4x2 Payload	508
SIMD8 Payload	511
Geometry Shader (GS) Stage	514
GS Stage Overview	514
State	514

Functions	515
Object Staging.....	515
Thread Request Generation	516
Object Vertex Ordering	516
Thread Execution	520
GS URB Entry	521
GS URB Entry - Output Vertex Count	522
GS Output Topologies.....	523
GS Output StreamID.....	523
Primitive Output	524
Statistics Gathering.....	524
Payloads.....	525
Thread Payload High-Level Layout	525
SIMD 4x2 Thread Payload.....	526
SIMD8 Thread Payload.....	533
Thread Request Generation	538
Thread Control Information.....	539
Thread Payload Generation	540
Fixed Payload Header.....	540
Extended Payload Header.....	543
Payload URB Data	543
Stream Output Logic (SOL) Stage.....	545
State	545
Functions	547
Input Buffering	547
Stream Output Function.....	551
Stream Output Buffers	552
Rendering Disable.....	553
Statistics	553
3D Pipeline Rasterization	554
Common Rasterization State	554
3D Pipeline – CLIP Stage Overview	554
Clip Stage – 3D Clipping	554

Fixed Function Clipper	555
Concepts.....	555
CLIP Stage Input	555
State.....	555
VUE Readback	557
VertexClipTest Function	557
Object Staging	561
Partial Object Removal.....	561
ClipDetermination Function.....	561
ClipMode State.....	564
NORMAL ClipMode	565
CLIP_ALL ClipMode	565
CLIP_NON_REJECT ClipMode.....	565
REJECT_ALL ClipMode.....	565
ACCEPT_ALL ClipMode	565
Object Pass-Through.....	565
Primitive Output	567
Other Functionality.....	567
Statistics Gathering.....	567
CL_INVOCATION_COUNT.....	567
3D Pipeline - Strips and Fans (SF) Stage	568
Inputs from CLIP	568
Attribute Setup/Interpolation Process.....	569
Attribute Setup/Interpolation Process	569
Outputs to WM.....	569
Primitive Assembly	570
Point List Decomposition	573
Line List Decomposition	575
Line Strip Decomposition.....	576
Triangle List Decomposition	578
Triangle Strip Decomposition.....	579
Triangle Fan Decomposition.....	580
Polygon Decomposition	581

Rectangle List Decomposition	581
Object Setup	582
Invalid Position Culling (Pre/Post-Transform)	582
Viewport Transformation	582
Destination Origin Bias	582
Point Rasterization Rule Adjustment	584
Drawing Rectangle Offset Application	585
Point Width Application	587
Rectangle Completion	588
Vertex XY Clamping and Quantization	589
Degenerate Object Culling	589
Triangle Orientation (Face) Culling	590
Scissor Rectangle Clipping	591
Viewport Extents Test	591
Line Rasterization	592
Zero-Width (Cosmetic) Line Rasterization	592
GIQ (Diamond) Sampling Rules – Legacy Mode	593
GIQ (Diamond) Sampling Rules – DX10 Mode	595
Non-Antialiased Wide Line Rasterization	597
Anti-Aliased Line Rasterization	597
SF Pipeline State Summary	599
3DSTATE_RASTER	599
3DSTATE_SF	599
Attribute Interpolation Setup	603
Attribute Swizzling	603
Interpolation Modes	604
Point Sprites	604
Barycentric Attribute Interpolation	605
Depth Offset	606
Other SF Functions	606
Statistics Gathering	606
Windower (WM) Stage	607
Overview	607

Inputs from SF to WM.....	607
Windower Pipelined State	608
3DSTATE_WM	608
3DSTATE_SAMPLE_MASK.....	612
Rasterization	617
Drawing Rectangle Clipping	617
Line Rasterization	617
Coverage Values for Anti-Aliased Lines.....	618
3DSTATE_AA_LINE_PARAMS	618
Line Stipple.....	619
Polygon (Triangle and Rectangle) Rasterization	619
Polygon Stipple	620
Multisampling	621
Multisample ModesState	622
Other WM Functions	628
Statistics Gathering.....	628
Pixel.....	629
Early Depth/Stencil Processing	629
Depth Offset	629
Early Depth Test/Stencil Test/Write.....	630
Software-Provided PS Kernel Info	631
Hierarchical Depth Buffer	632
Depth Buffer Clear	633
Depth Buffer Resolve	636
Hierarchical Depth Buffer Resolve.....	637
Optimized Depth Buffer Clear and/or Stencil Buffer Clear	638
Optimized Depth Buffer Resolve.....	639
Optimized Hierarchical Depth Buffer Resolve	639
Separate Stencil Buffer.....	639
DepthStencil Buffer State	640
Pixel Shader Thread Generation	641
3DSTATE_PS.....	641
Pixel Grouping (Dispatch Size) Control	643

Multisampling Effects on Pixel Shader Dispatch	645
MSDISPMODE_PERPIXEL Thread Dispatch.....	645
MSDISPMODE_PERSAMPLE Thread Dispatch.....	645
PS Thread Payload for Normal Dispatch	649
PS Thread Payload for Normal Dispatch.....	649
Pixel Backend	661
Color Calculator (Output Merger)	661
Overview	661
Alpha Coverage.....	662
Alpha Test.....	662
Depth Coordinate Offset.....	663
Stencil Test.....	664
Depth Test.....	665
Pre-Blend Color Clamping	666
Pre-Blend Color Clamping When Blending is Disabled	667
Pre-Blend Color Clamping When Blending is Enabled.....	667
Color Buffer Blending	667
Post-Blend Color Clamping.....	670
Dithering.....	670
Logic Ops.....	671
Buffer Update.....	672
Stencil Buffer Updates	672
Depth Buffer Updates	673
Color Gamma Correction.....	673
Color Buffer Updates.....	674
Pixel Pipeline State Summary	674
COLOR_CALC_STATE	674
3DSTATE_BLEND_STATE_POINTERS	674
3DSTATE_DEPTH_STENCIL_STATE_POINTERS.....	674
COLOR_CALC_STATE	674
DEPTH_STENCIL_STATE	674
BLEND_STATE.....	674
CC_VIEWPORT.....	675

Other Pixel Pipeline Functions	675
Statistics Gathering	675
MCS Buffer for Render Target(s)	675
Render Target Fast Clear	678
Render Target Resolve	678
Media GPGPU Pipeline	680
Programming the GPGPU Pipeline	680
GPGPU Thread Limits	680
GPGPU Commands	681
GPGPU Command Workarounds	681
GPGPU Indirect Thread Dispatch	682
GPGPU Context Switch	683
GPGPU Context Switch	684
Media GPGPU Payload Limitations	687
Synchronization of the Media/GPGPU Pipeline	688
Mode of Operations	689
GPGPU Thread R0 Header	689
GPGPU Mode	691
Automatic Thread Generation	691
Thread Payload	692
Execution Masks	693
URB Management	695
Indirect Payload Storage	695
MEDIA_OBJECT_GRPID	698
Starting Offset for a Thread Group ID	698
Thread Group Tracking	699
Shared Local Memory Allocation	700
Software Managed Shared Local Memory	701
Automatic Barrier Management	701
Dispatch Payload	702
Generic Media	702
Product Evolution	704
Media and General Purpose Pipeline	705

Introduction	705
Terminology	706
Hardware Feature Map in Products.....	707
Media Pipeline Overview	708
Generic Mode.....	709
Programming Media Pipeline.....	710
Command Sequence	710
Command Sequence.....	710
Parameterized Media Walker	713
Walker Parameter Description	714
Basic Parameters for the Local Loop	715
MbAff-Like Special Case in Local Loop.....	717
Global Loop	718
Walker Algorithm Description.....	719
Barriers and Shared Local Memory	723
Flexible Dispatch of Local Loop.....	724
Scoreboard Control.....	725
AVC-Style Dependency Example	726
Interface Descriptor Selection	728
VC1-Style Dependency Example	729
Multiple Slice Considerations.....	730
Interrupt Latency.....	730
Thread Spawner Unit.....	731
Root Threads and Child Threads	732
Root Threads	732
URB Handles	733
Root to Child Responsibilities.....	733
Multiple Simultaneous Roots	733
Synchronized Root Threads.....	734
Deadlock Prevention	735
Child Thread Life Cycle	736
Arbitration between Root and Child Threads.....	736
Persistent Root Thread	737

Media State Model	738
Media State and Primitive Commands	739
Media State and Primitive Command Workarounds	740
Media Messages	741
Thread Payload Messages	741
Generic Mode Root Thread	741
Root Thread from MEDIA_OBJECT_PRT	743
Root Thread from MEDIA_OBJECT_WALKER	744
MEDIA_OBJECT_GRPID and MEDIA_OBJECT_WALKER with Groups Payload	745
Thread Spawn Message	746
Message Descriptor	747
Message Payload	748
L3 Cache and URB	750
L3 Cache and URB	750
Overview	751
L3 Cache Configuration	751
Block(s) Overview	752
Size of L3 for 8th Generation	754
ECC Protection	754
L3 Cache Theory of Operation	754
Memory Object Control State on Cacheability	755
L3 Allocation and Programming	755
Non-SLM Mode Allocation	756
SLM Mode Allocation	758
L3 Cache Error Protection	759
Dynamic Parity Feature for GFX L3 Cache	759
Feature Definition	759
Hardware and Software Flows	760
Parity Generation & Detection	760
Correction Using Parity Error data and Redundant Rows	760
Number of Corrections	760
Summary	761
Sub-banks with more than two persistent parity error rows	761

Interrupt Enabling	761
Clearing the Error Reporting Registers	761
Managing H/W Context via WABB	762
EU Overview	763
CoIssue/Dual Issue:	763
Thread scheduling:	764
Primary Usage Models	765
AOS and SOA Data Structures	765
SIMD4 Mode of Operation	767
SIMD4x2 Mode of Operation	768
SIMD16 Mode of Operation	770
SIMD8 Mode of Operation	773
Messages	774
Message Payload Containing a Header	776
Writebacks	777
Message Delivery Ordering Rules	778
Execution Mask and Messages	779
End-Of-Thread (EOT) Message	780
Performance	781
Message Description Syntax	782
Message Errors	783
Registers and Register Regions	785
Register Files	785
GRF Registers	786
ARF Registers	787
ARF Registers Overview	787
Access Granularity	788
Null Register	789
Address Register	790
Accumulator Registers	794
Flag Register	799
Channel Enable Register	801
Message Control Registers	802

Example	802
SP Register	803
State Register	805
Control Register	811
Notification Registers	817
IP Register	819
TDR Registers	820
Performance Registers	822
Flow Control Registers	824
Immediate	828
Region Parameters	829
Region Addressing Modes	833
Direct Register Addressing	833
Register-Indirect Register Addressing with a 1x1 Index Region	835
Register-Indirect Register Addressing with a Vx1 Index Region	836
Register-Indirect Register Addressing with a VxH Index Region	837
Access Modes	838
Execution Data Type	839
Register Region Restrictions	840
Destination Operand Description	846
Destination Region Parameters	846
SIMD Execution Control	847
Predication	847
No Predication	848
Predication with Horizontal Combination	849
Predication with Vertical Combination	851
End of Thread	852
Assigning Conditional Flags	853
Destination Hazard	856
Non-present Operands	857
Instruction Prefetch	857
ISA Introduction	858
Introducing the Execution Unit	859

EU Terms and Acronyms	862
Execution Units (EUs).....	867
EU Changes by Processor Generation	868
EU Notation.....	871
Execution Environment	872
EU Data Types	872
Fundamental Data Types	872
Numeric Data Types	874
Integer Numeric Data Types.....	874
Floating-Point Numeric Data Types.....	876
Packed Signed Half-Byte Integer Vector	878
Packed UnSigned Half-Byte Integer Vector.....	879
Packed Restricted Float Vector	880
Floating Point Modes	882
IEEE Floating Point Mode.....	882
Partial Listing of Honored IEEE-754 Rules	882
Complete Listing of Deviations or Additional Requirements vs IEEE-754	883
Min Max of Floating Point Numbers	884
Alternative Floating Point Mode	886
Floating-Point Support	888
Floating-Point Types and Values	888
Not a Number (NaN) Formats	889
Floating-Point Rounding Modes.....	890
Floating-Point Operations and Precision.....	891
Single Precision Floating-Point Rounding to Integral Values	891
Floating-Point to Integer Conversion.....	892
Integer to Floating-Point Conversion.....	893
Floating-Point Min/Max Operations.....	893
IEEE Floating-Point Exceptions.....	894
Signaling Floating-Point Exceptions	894
Invalid Operation Exception.....	895
Division by Zero Exception	896
Overflow Exception.....	896

Underflow Exception	897
Inexact Exception	898
Floating-Point Compare Operations	898
Type Conversion.....	906
Float to Integer.....	906
Integer to Integer with Same or Higher Precision	906
Integer to Integer with Lower Precision	907
Integer to Float.....	907
Double Precision Float to Single Precision Float	908
Single Precision Float to Double Precision Float	908
Exceptions.....	909
Exception-Related Architecture Registers.....	910
System Routine	911
Invoking the System Routine.....	911
Returning to the Application Thread.....	912
System IP (SIP).....	913
System Routine Register Space	913
Conditional Instructions Within the System Routine	914
Use of NoDDClr	914
Exception Descriptions	916
Illegal Opcode	916
Undefined Opcodes	916
Software Exception	916
Context Save and Restore.....	916
Events That Do Not Generate Exceptions	918
Illegal Instruction Format	918
Malformed Message	918
GRF Register Out of Bounds	918
Hung Thread	918
Instruction Fetch Out of Bounds	918
FPU Math Errors.....	918
Computational Overflow	919
System Routine Example.....	920

Instruction Set Summary	923
Instruction Set Characteristics	923
SIMD Instructions and SIMD Width	923
Instruction Operands and Register Regions.....	923
Instruction Execution	924
Instruction Formats	925
Native Instruction Layouts.....	928
Instruction Fields	933
EU Compact Instructions	948
EU Compact Instruction Format	948
EU Instruction Compaction Tables	953
Opcode Encoding	958
Move and Logic Instructions	959
Flow Control Instructions	961
Miscellaneous Instructions	962
Parallel Arithmetic Instructions.....	963
Vector Arithmetic Instructions.....	964
Special Instructions.....	965
Native Instruction BNF.....	966
Instruction Groups	966
Destination Register.....	968
Source Register	969
Address Registers.....	970
Register Files and Register Numbers	971
Relative Location and Stack Control.....	972
Regions.....	972
Types	972
Write Mask.....	973
Swizzle Control	973
Immediate Values.....	973
Predication and Modifiers.....	974
Instruction Options.....	975
Instruction Set Summary Tables	976

Accumulator Restrictions	979
Instruction Set Reference	982
Conventions	982
Pseudo Code Format	982
General Macros and Definitions	982
Evaluate Write Enable	983
EUISA Instructions	984
Round Instructions	987
rndd – Round Down	987
rnde – Round to Nearest or Even	988
rndu – Round Up	989
rndz – Round to Zero	990
math – Extended Math Function	991
INV - Inverse	991
LOG – Logarithm	992
EXP - Exponent	992
SQRT - Square Root	993
RSQ - Reciprocal Square Root	993
POW - Power Function	994
SIN - SINE	995
COS - COSINE	996
INT DIV - Integer Divide	996
INVM/RSQRTM	997
Send Message	1000
Send Message	1000
EUISA Structures	1005
EUISA Enumerations	1006
EU Programming Guide	1007
Assembler Pragma Declarations	1007
Defaults and Defines	1008
Example Pragma Usages	1009
Assembly Programming Guideline	1011
Usage Examples	1012

Vector Immediate.....	1012
Supporting DirectX 10 Pixel Shader Indexing	1012
Supporting OpenGL Vertex Shader Instruction SWZ	1013
Destination Mask for DP4 and Destination Dependency Control	1014
Null Register as the Destination.....	1015
Use of LINE Instruction	1016
Mask for SEND Instruction	1018
Channel Enables for Extended Math Unit	1018
Channel Enables for Scratch Memory	1019
Flow Control Instructions	1022
Execution Masking.....	1023
Branching.....	1023
Fast-If.....	1024
Cascade Branching.....	1025
Compound Branches.....	1026
Looping.....	1027
Indexed Jump.....	1031

Registers in Render Engine

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. Unless noted otherwise within the description of the bit, any mode bit must only be updated following a flush to ensure the pipeline is idle.

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

Predication

Predicate Render Registers

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 choose to execute or skip a command during execution time.

Below is a table of command(s) that are allowed to be programmed when the MI_SET_PREDICATE PREDICATE_ENABLE field allows any type of predication. If a command is not in the table, then it will have undefined behavior when executed with predication enabled:

MI_SET_PREDICATE

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
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

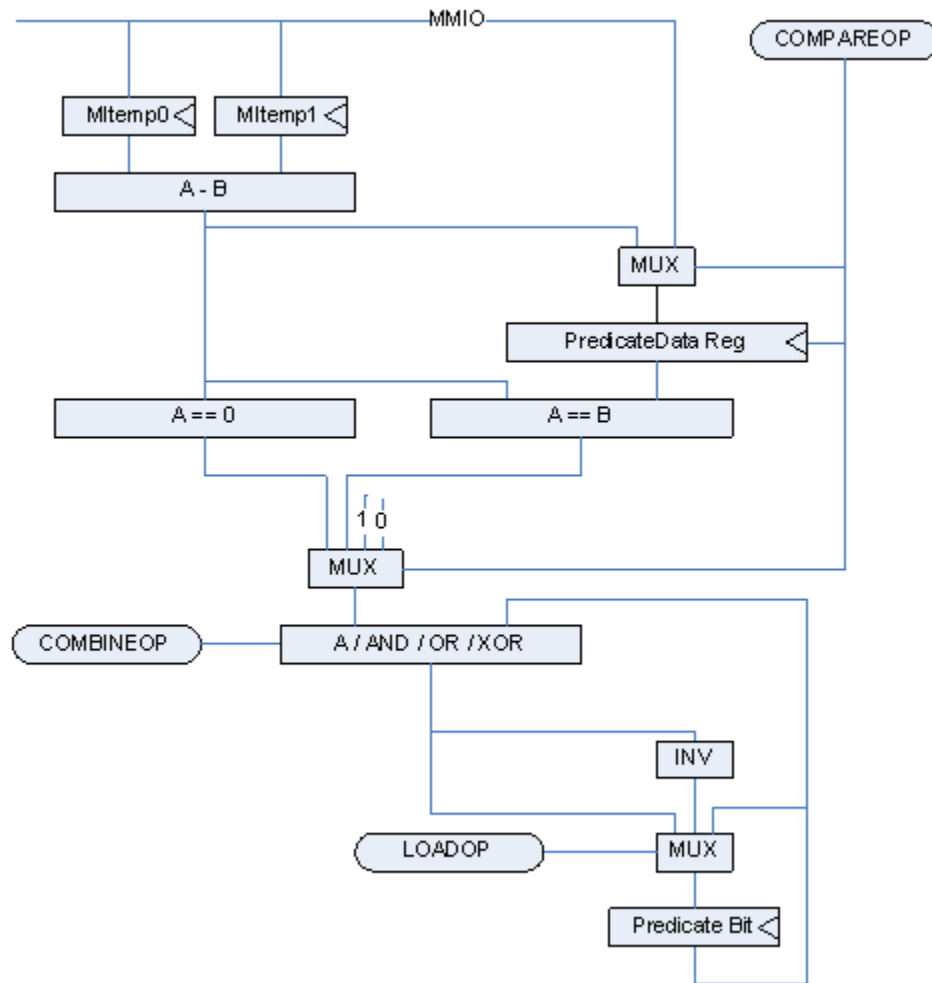
Predicated Rendering Support in HW

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 Mltemp0, Mltemp1, 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.

Registers in Render Engine

This section 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. Unless noted otherwise within the description of the bit, any mode bit must only be updated following a flush to ensure the pipeline is idle.

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

Context Save Registers

VF Instance Count Registers

VF Instance Count Register Set		
Register Type:	MMIO_VF	
Address:	08300h - 08384h	
Default Value:	0000 0000h	
Access:	RO	
Size:	1088 bits	
Description:	<p>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.</p> <p>These are saved as part of render context.</p>	
DWord	Bits	Description
0	31:0	Index Count 0. Index Count value for Element 0. Format: U32
1	31:0	Index Count 1. Index Count value for Element 1. Format: U32
...	31:0	...
33	31:0	Index Count 33. Index Count value for Element 33. Format: U32

Mode and Misc Ctrl Registers

This section contains various registers for controls and modes.

MI_MODE - Render Mode Register for Software Interface

FF_MODE - Thread Mode Register

GFX_MODE - Graphics Mode Register

GT_MODE - GT Mode Register

SAMPLER_MODE - SAMPLER Mode Register

CACHE_MODE_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.

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.

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

SO_NUM_PRIMS_WRITTEN[0:3] - Stream Output Num Primitives Written Counter

SO_PRIM_STORAGE_NEEDED[0:3] - Stream Output Primitive Storage Needed Counters

SO_WRITE_OFFSET[0:3] - Stream Output Write Offsets

WMHWCLRVAL - Window Hardware Generated Clear Value

CS_CTX_TIMESTAMP- CS Context Timestamp Count:

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

CS_CTX_TIMESTAMP - CS Context Timestamp Count

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

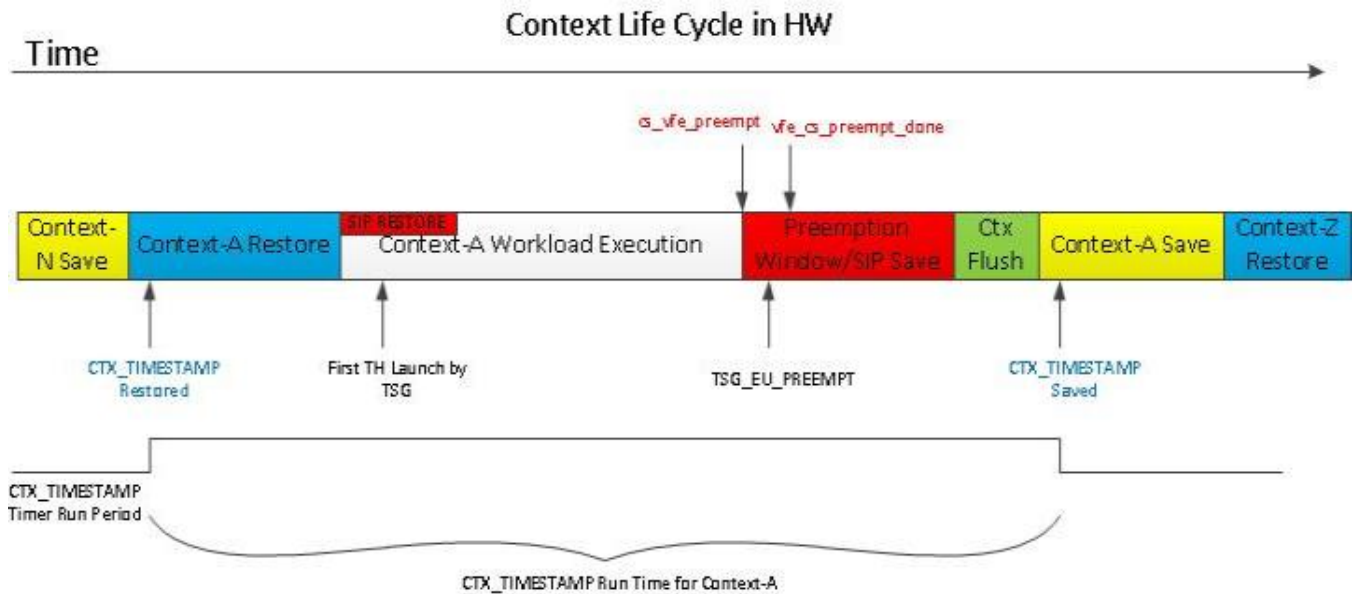


Fig: CTX_TIMESTAMP functionality during context execution

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

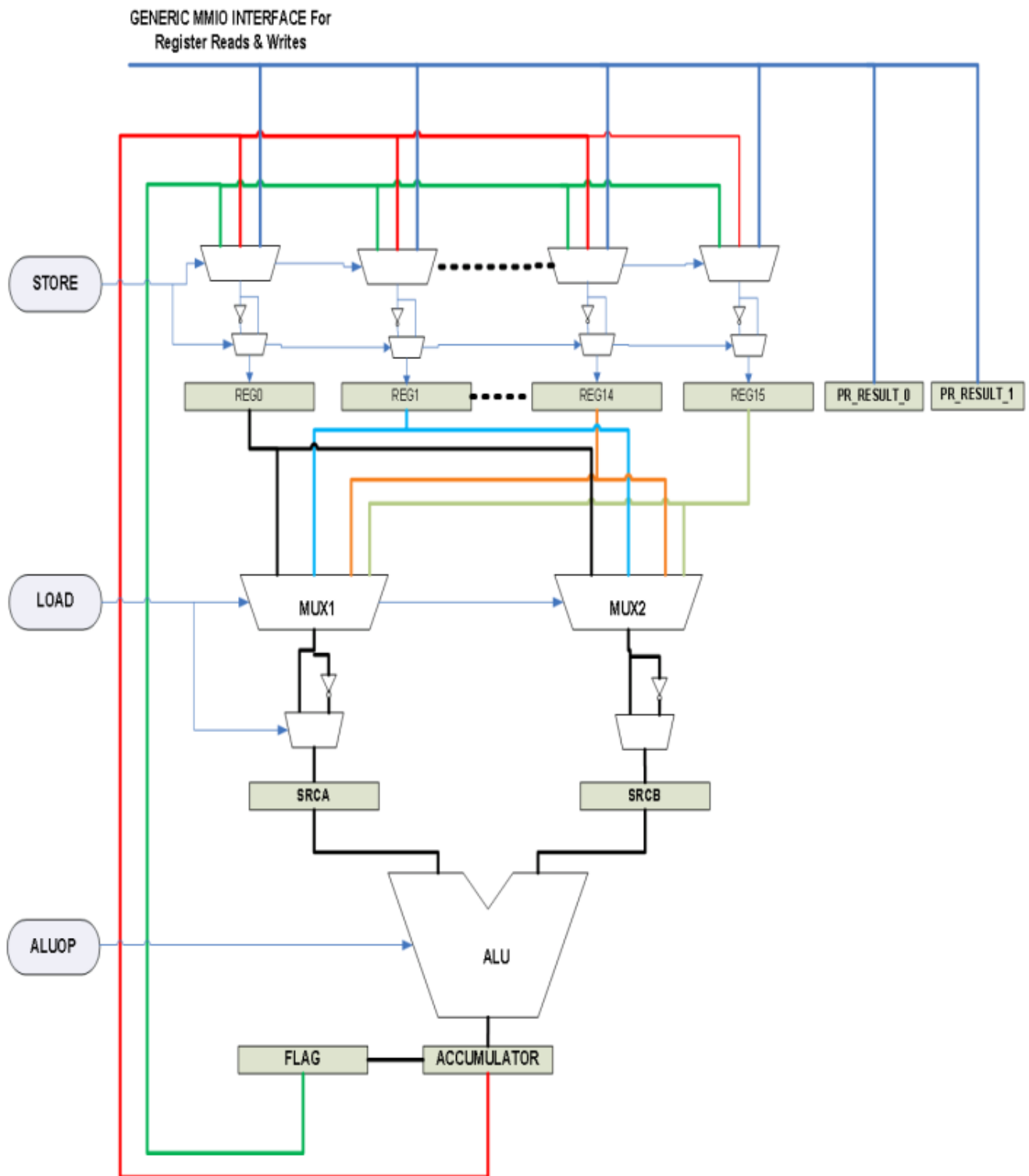
CS ALU Programming and Design

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.

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.

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 and GPR (R0, R1 ..R15). 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 ACCUMULATOR 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 (ACCUMULATOR 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

ALU OPCODE	OPCODE ENCODING
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
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 - CS General Purpose Register

Memory Interface Commands for Rendering Engine

MI_SET_CONTEXT

MI_TOPOLOGY_FILTER

MI_URB_CLEAR

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**

STATE_PREFETCH

STATE_SIP

STATE_BASE_ADDRESS

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

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

- 3DSTATE_PIPELINE_POINTERS
- 3DSTATE_BINDING_TABLE_POINTERS
- MEDIA_STATE_POINTERS

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

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 is used to effect the synchronization described above. 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

Hardware supports up to 32 pending PIPE_CONTROL flushes.

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

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

PIPE_CONTROL

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.
LRI Post Sync Operation	23	Requires stall bit ([20] of DW1) set.
LRI Post Sync Operation	23	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
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'.
Post Sync Op	15:14	"CS Stall" must always be set in all PIPE_CONTROL commands having "Post-Sync Operation" set to "Report PS Depth Count" or "Report Time Stamp" when user wishes to set "Post-Sync Operation" bit in any of the MI_ATOMIC or MI_SEMAPHORE_SIGNAL commands programmed.
Post Sync Op	15:14	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
Notify En	8	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.

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	Bits	Restrictions
Depth Stall	13	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
Render Target Cache Flush	12	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
Depth Cache Flush	0	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
Stall Pixel Scoreboard	1	No Restriction.
DC Flush Enable	5	Requires stall bit ([20] of DW) set for all GPGPU and Media Workloads.
Inst invalidate	11	No Restriction.
Tex invalidate	10	No Restriction.
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	<p>[All Stepping][All SKUs]:</p> <p>One of the following must also be set:</p> <ul style="list-style-type: none"> • Render Target Cache Flush Enable ([12] of DW1) • Depth Cache Flush Enable ([0] of DW1) • Stall at Pixel Scoreboard ([1] of DW1) • Depth Stall ([13] of DW1) • Post-Sync Operation ([13] of DW1) • DC Flush Enable ([5] of DW1)

Render Logical Context Data

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.

Overall Context Layout

Context Layout

When Execlists are enabled, the Context Image for the rendering engine consists of 20 4K pages:

Per-Process HW Status Page (4K)
Register State Context

When Execlists are disabled, the context image doesn't consist the Per-Process HW status page.

Register State context is explained in detail in "Register State Context" Section.

Register State Context

This section describes programming requirements for the Register State Context.

Programming Note	
Context:	Register State Context.
<ul style="list-style-type: none"> All the MMIO registers part of the "Register 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 "Register 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. MMIO reads or writes to any of the registers listed in the "Register State Context image" subsections through HOST/IA MMIO interface for debug purposes must follow the steps below: <ul style="list-style-type: none"> SW should set the Force Wakeup bit to prevent GT from entering C6. Write 0x2050[31:0] = 0x00010001 (disable sequence). Disable IDLE messaging in CS (Write 0x2050[31:0] = 0x00010001). Poll/Wait for register bits of <u>0x22AC[6:0]</u> turn to 0x30 value. Read/Write to desired MMIO registers. Enable IDLE messaging in CS (Write 0x2050[31:0] = 0x00010000). Force Wakeup bit should be reset to enable C6 entry. 	

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 (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
NOOP		CS	1	0	0	0
Load_Register_Immediate header	0x1100_101B	CS	1	0001	0001	0001
Context Control	0x2244	CS	2	0002	0002	0002
Ring Head Pointer Register	0x2034	CS	2	0004	0004	0004
Ring Tail Pointer Register	0x2030	CS	2	0006	0006	0006
RING_BUFFER_START	0x2038	CS	2	0008	0008	0008
RING_BUFFER_CONTROL	0x203C	CS	2	000A	000A	000A
Batch Buffer Current Head Register (UDW)	0x2168	CS	2	000C	000C	000C
Batch Buffer Current Head Register	0x2140	CS	2	000E	000E	000E
Batch Buffer State Register	0x2110	CS	2	0010	0010	0010
SECOND_BB_ADDR_UDW	0x211C	CS	2	0012	0012	0012
SECOND_BB_ADDR	0x2114	CS	2	0014	0014	0014
SECOND_BB_STATE	0x2118	CS	2	0016	0016	0016
BB_PER_CTX_PTR	0x21C0	CS	2	0018	0018	0018
RCS_INDIRECT_CTX	0x21C4	CS	2	001A	001A	001A
RCS_INDIRECT_CTX_OFFSET	0x21C8	CS	2	001C	001C	001C
NOOP		CS	2	001E	001E	001E
NOOP		CS	1	0020	0020	0020
Load_Register_Immediate header	0x1100_1011	CS	1	0021	0021	0021
CTX_TIMESTAMP	23A8	CS	2	0022	0022	0022
PDP3_UDW	228C	CS	2	0024	0024	0024
PDP3_LDW	2288	CS	2	0026	0026	0026
PDP2_UDW	2284	CS	2	0028	0028	0028
PDP2_LDW	2280	CS	2	002A	002A	002A

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
PDP1_UDW	227C	CS	2	002C	002C	002C
PDP1_LDW	2278	CS	2	002E	002E	002E
PDP0_UDW	2274	CS	2	0030	0030	0030
PDP0_LDW	2270	CS	2	0032	0032	0032
NOOP		CS	12	0034	0034	0034
NOOP		CS	1	0040	0040	0040
Load_Register_Immediate header	0x1100_0001	CS	1	0041	0041	0041
R_PWR_CLK_STATE	20C8	CS	2	0042	0042	0042
GPGPU_CSR_BASE_ADDRESS		CS	3	0044	0044	0044
NOOP		CS	9	0047	0047	0047
NOOP		CS	5	0050	0050	0050
Load_Register_Immediate header	0x1100_10D7	CS	1	0055	0055	0055
EXCC	0x2028	CS	2	0056	0056	0056
MI_MODE	0x209C	CS	2	0058	0058	0058
INSTPM	0x20C0	CS	2	005A	005A	005A
PR_CTR_CTL	0x2178	CS	2	005C	005C	005C
PR_CTR_THRSH	0x217C	CS	2	005E	005E	005E
IA_VERTICES_COUNT	0x2310	CS	4	0062	0062	0062
IA_PRIMITIVES_COUNT	0x2318	CS	4	0066	0066	0066
VS_INVOCATION_COUNT	0x2320	CS	4	006A	006A	006A
HS_INVOCATION_COUNT	0x2300	CS	4	006E	006E	006E
DS_INVOCATION_COUNT	0x2308	CS	4	0072	0072	0072
GS_INVOCATION_COUNT	0x2328	CS	4	0076	0076	0076
GS_PRIMITIVES_COUNT	0x2330	CS	4	007A	007A	007A
CL_INVOCATION_COUNT	0x2338	CS	4	007E	007E	007E
CL_PRIMITIVES_COUNT	0x2340	CS	4	0082	0082	0082
PS_INVOCATION_COUNT_0	0x22C8	CS	4	0086	0086	0086
PS_DEPTH_COUNT_0	0x22D8	CS	4	008A	008A	008A
TIMESTAMP Register (LSB)	0x2358	CS	2	008E	008E	008E
GPUGPU_DISPATCHDIMX	0x2500	CS	2	0092	0092	0092
GPUGPU_DISPATCHDIMY	0x2504	CS	2	0094	0094	0094
GPUGPU_DISPATCHDIMZ	0x2508	CS	2	0096	0096	0096
MI_PREDICATE_SRC0	0x2400	CS	2	0098	0098	0098
MI_PREDICATE_SRC0	0x2404	CS	2	009A	009A	009A
MI_PREDICATE_SRC1	0x2408	CS	2	009C	009C	009C

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
MI_PREDICATE_SRC1	0x240C	CS	2	009E	009E	009E
MI_PREDICATE_DATA	0x2410	CS	2	00A0	00A0	00A0
MI_PREDICATE_DATA	0x2414	CS	2	00A2	00A2	00A2
MI_PRED_RESULT	0x2418	CS	2	00A4	00A4	00A4
3DPRIM_END_OFFSET	0x2420	CS	2	00A6	00A6	00A6
3DPRIM_START_VERTEX	0x2430	CS	2	00A8	00A8	00A8
3DPRIM_VERTEX_COUNT	0x2434	CS	2	00AA	00AA	00AA
3DPRIM_INSTANCE_COUNT	0x2438	CS	2	00AC	00AC	00AC
3DPRIM_START_INSTANCE	0x243C	CS	2	00AE	00AE	00AE
3DPRIM_BASE_VERTEX	0x2440	CS	2	00B0	00B0	00B0
GPGPU_THREADS_DISPATCHED	0x2290	CS	4	00B2	00B2	00B2
PS_INVOCATION_COUNT_1	0x22F0	CS	4	00B6	00B6	00B6
PS_DEPTH_COUNT_1	0x22F8	CS	4	00BA	00BA	00BA
BB_START_ADDR_UDW	0x2170	CS	2	00BE	00BE	00BE
BB_START_ADDR	0x2150	CS	2	00C0	00C0	00C0
BB_ADD_DIFF	0x2154	CS	2	00C2	00C2	00C2
BB_OFFSET	0x2158	CS	2	00C4	00C4	00C4
RS_PREEMPT_STATUS	0x215C	CS	2	00C6	00C6	00C6
CTX_SEMA_REG	0x24B4	CS	4	00C8	00C8	00C8
PRODUCE_COUNT_BTP	0x2480	CS	2	00CC	00CC	00CC
PRODUCE_COUNT_DX9_CONSTANTS	0x2484	CS	2	00CE	00CE	00CE
PRODUCE_COUNT_GATHER_CONSTANTS	0x248C	CS	2	00D0	00D0	00D0
PARSED_COUNT_BTP	0x2490	CS	2	00D2	00D2	00D2
PARSED_COUNT_DX9_CONSTANTS	0x2494	CS	2	00D4	00D4	00D4
MI_PREDICATE_RESULT_1	0x241C	CS	2	00D6	00D6	00D6
CS_GPR (1-16)	0x2600	CS	64	00D8	00D8	00D8
MI_TAGADDR	0x2194	CS	4	0118	0118	0118
MI_TAGDATA	0x219C	CS	4	011C	011C	011C
OA_CTX_CONTROL	0x2360	CS	2	0120	0120	0120
PS_INVOCATION_COUNT_2	0x2448	CS	4	0124	0124	0124
PS_DEPTH_COUNT_2	0x2450	CS	4	0128	0128	0128
RS_PREEMPT_STATUS_UDW	0x2174	CS	2	012C	012C	012C
NOOP		CS	10	012E	012E	012E
MI_TOPOLOGY_FILTER		CS	1	0138	0138	0138
NOOP		CS	2	0139	0139	0139

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
PIPELINE_SELECT		CS	1	013B	013B	013B
STATE_BASE_ADDRESS		CS	16	013C	013C	013C
3DSTATE_PUSH_CONSTANT_ALLOC_VS		CS	2	014C	014C	014C
3DSTATE_PUSH_CONSTANT_ALLOC_HS		CS	2	014E	014E	014E
3DSTATE_PUSH_CONSTANT_ALLOC_DS		CS	2	0150	0150	0150
3DSTATE_PUSH_CONSTANT_ALLOC_GS		CS	2	0152	0152	0152
3DSTATE_PUSH_CONSTANT_ALLOC_PS		CS	2	0154	0154	0154
3DSTATE_BINDING_TABLE_POOL_ALLOC		CS	4	0156	0156	0156
3DSTATE_GATHER_POOL_ALLOC		CS	4	015A	015A	015A
3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC		CS	4	015E	015E	015E
MI_RS_CONTROL		CS	1	0162	0162	0162
MI_URB_ATOMIC_ALLOC		CS	1	0163	0163	0163
NOOP		CS	12	0164	0164	0164
3DSTATE_VS		SVG	9	0170	0170	0170
3DSTATE_BINDING_TABLE_POINTERS_VS		SVG	2	0179	0179	0179
3DSTATE_SAMPLER_STATE_POINTERS_VS		SVG	2	017B	017B	017B
3DSTATE_URB_VS		SVG	2	017D	017D	017D
3DSTATE_HS		SVG	9	017F	017F	017F
3DSTATE_BINDING_TABLE_POINTERS_HS		SVG	2	0188	0188	0188
3DSTATE_SAMPLER_STATE_POINTERS_HS		SVG	2	018A	018A	018A
3DSTATE_URB_HS		SVG	2	018C	018C	018C
3DSTATE_TE		SVG	4	018E	018E	018E
3DSTATE_DS		SVG	9	0192	0192	0192
3DSTATE_BINDING_TABLE_POINTERS_DS		SVG	2	019B	019B	019B
3DSTATE_SAMPLER_STATE_POINTERS_DS		SVG	2	019D	019D	019D
3DSTATE_URB_DS		SVG	2	019F	019F	019F
3DSTATE_GS		SVG	10	01A1	01A1	01A1
3DSTATE_BINDING_TABLE_POINTERS_GS		SVG	2	01AB	01AB	01AB
3DSTATE_SAMPLER_STATE_POINTERS_GS		SVG	2	01AD	01AD	01AD
3DSTATE_URB_GS		SVG	2	01AF	01AF	01AF
3DSTATE_STREAMOUT		SVG	5	01B1	01B1	01B1
3DSTATE_CLIP		SVG	4	01B6	01B6	01B6
3DSTATE_SF		SVG	4	01BA	01BA	01BA
3DSTATE_SCISSOR_STATE_POINTERS		SVG	2	01BE	01BE	01BE

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
3DSTATE_VIEWPORT_STATE_POINTERS_C L_SF		SVG	2	01C0	01C0	01C0
3DSTATE_RASTER		SVG	5	01C2	01C2	01C2
3DSTATE_WM_HZ_OP		SVG	5	01C7	01C7	01C7
3DSTATE_MULTISAMPLE		SVG	2	01CC	01CC	01CC
3DSTATE_CONSTANT_VS		SVG	11	01CE	01CE	01CE
3DSTATE_CONSTANT_HS		SVG	11	01D9	01D9	01D9
3DSTATE_CONSTANT_DS		SVG	11	01E4	01E4	01E4
3DSTATE_CONSTANT_GS		SVG	11	01EF	01EF	01EF
3DSTATE_DRAW_RECTANGULAR		SVG	4	01FA	01FA	01FA
3DSTATE_SW_TE_BASE_ADDRESS		SVG	3	01FE	01FE	01FE
Load_Register_Immediate header	0x1100_1001	SVG	1	0201	0201	0201
FF_PERF_REG	0x6b1c	SVG	2	0202	0202	0202
NOOP		SVG	12	0204	0204	0204
3DSTATE_WM		SVL	2	0210	0210	0210
3DSTATE_VIEWPORT_STATE_POINTER_CC		SVL	2	0212	0212	0212
3DSTATE_CC_STATE_POINTERS		SVL	2	0214	0214	0214
3DSATE_WM_SAMPLEMASK		SVL	2	0216	0216	0216
3DSTATE_DEPTH_STENCIL		SVL	3	0218	0218	0218
3DSTATE_WM_CHROMAKEY		SVL	2	021B	021B	021B
3DSTATE_DEPTH_BUFF		SVL	8	021D	021D	021D
3DSTATE_HIZ_DEPTH_BUFF		SVL	5	0225	0225	0225
3DSTATE_STC_DEPTH_BUFF		SVL	5	022A	022A	022A
3DSTATE_CLEAR_PARAMS		SVL	3	022F	022F	022F
3DSTATE_SBE		SVL	4	0232	0232	0232
3DSTATE_SBE_SWIZ		SVL	11	0236	0236	0236
3DSTATE_PS		SVL	12	0241	0241	0241
3DSTATE_BINDING_TABLE_POINTERS_PS		SVL	2	024D	024D	024D
STATE_SAMPLER_STATE_POINTERS_PS		SVL	2	024F	024F	024F
3DSTATE_BLEND_STATE_POINTERS		SVL	2	0251	0251	0251
3DSTATE_PS_EXTRA		SVL	2	0253	0253	0253
3DSTATE_PS_BLEND		SVL	2	0255	0255	0255
3DSTATE_CONSTANT_PS		SVL	11	0257	0257	0257
STATE_SIP		SVL	3	0262	0262	0262
3DSTATE_SAMPLE_PATTERN		SVL	9	0265	0265	0265

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
Load_Register_Immediate header	0x1100_1019	SVL	1	026E	026E	026E
Cache_Mode_0	0x7000	SVL	2	026F	026F	026F
Cache_Mode_1	0x7004	SVL	2	0271	0271	0271
GT_MODE	0x7008	SVL	2	0273	0273	0273
FBC_RT_BASE_ADDR_REGISTER	0x7020	SVL	2	0279	0279	0279
FBC_RT_BASE_ADDR_REGISTER_UPPER	0x7024	SVL	2	027B	027B	027B
SAMPLER_MODE	0x7028	SVL	2	027D	027D	027D
L3_Config	0x7034	SVL	2	0287	0287	0287
NOOP		SVL	23	0289	0289	0289
NOOP		TDL	1	02A0	02A0	02A0
Load_Register_Immediate header	0x1100_1039	TDL	1	02A1	02A1	02A1
TD_CTL	E400	TDL	2	02A2	02A2	02A2
TD_CTL2	E404	TDL	2	02A4	02A4	02A4
TD_VF_VS_EMSK	E408	TDL	2	02A6	02A6	02A6
TD_GS_EMSK	E40C	TDL	2	02A8	02A8	02A8
TD_WIZ_EMSK	E410	TDL	2	02AA	02AA	02AA
TD_TS_EMSK	E428	TDL	2	02AC	02AC	02AC
TD_HS_EMSK	E4B0	TDL	2	02AE	02AE	02AE
TD_DS_EMSK	E4B4	TDL	2	02B0	02B0	02B0
EU_PERF_CNT_CTL0	E458	TDL	2	02CE	02CE	02CE
EU_PERF_CNT_CTL1	E558	TDL	2	02D0	02D0	02D0
EU_PERF_CNT_CTL2	E658	TDL	2	02D2	02D2	02D2
EU_PERF_CNT_CTL3	E758	TDL	2	02D4	02D4	02D4
EU_PERF_CNT_CTL4	E45C	TDL	2	02D6	02D6	02D6
EU_PERF_CNT_CTL5	E55C	TDL	2	02D8	02D8	02D8
EU_PERF_CNT_CTL6	E65C	TDL	2	02DA	02DA	02DA
NOOP		TDL	4	02DC	02DC	02DC
NOOP		WM	1	02E0	02E0	02E0
Load_Register_Immediate header	0x1100_1005	WM	1	02E1	02E1	02E1
WMHWCLRVAL	0x5524	WM	2	02E6	02E6	02E6
3DSTATE_POLY_STIPPLE_PATTERN		WM	33	02E8	02E8	02E8
3DSTATE_AA_LINE_PARAMS		WM	3	0309	0309	0309
3DSTATE_POLY_STIPPLE_OFFSET		WM	2	030C	030C	030C
3DSTATE_LINE_STIPPLE		WM	3	030E	030E	030E
NOOP		WM	15	0311	0311	0311

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
NOOP		SC	1	0320	0320	0320
Load_Register_Immediate header	0x1100_1005	SC	1	0321	0321	0321
3DSTATE_MONOFILTER_SIZE		SC	2	0328	0328	0328
3DSTATE_CHROMA_KEY		SC	16	032A	032A	032A
NOOP		SC	6	033A	033A	033A
NOOP		DM	1	0340	0340	0340
3DSTATE_SAMPLER_PALETTE_LOAD0		DM	257	0341	0341	0341
NOOP		DM	1	0442	0442	0442
3DSTATE_SAMPLER_PALETTE_LOAD1		DM	257	0443	0443	0443
NOOP		DM	12	0544	0544	0544
NOOP		SOL	1	0550	0550	0550
Load_Register_Immediate header	0x1100_1027	SOL	1	0551	0551	0551
SO_NUM_PRIMS_WRITTEN0	0x5200	SOL	4	0552	0552	0552
SO_NUM_PRIMS_WRITTEN1	0x5208	SOL	4	0556	0556	0556
SO_NUM_PRIMS_WRITTEN2	0x5210	SOL	4	055A	055A	055A
SO_NUM_PRIMS_WRITTEN3	0x5218	SOL	4	055E	055E	055E
SO_PRIM_STORAGE_NEEDED0	0x5240	SOL	4	0562	0562	0562
SO_PRIM_STORAGE_NEEDED1	0x5248	SOL	4	0566	0566	0566
SO_PRIM_STORAGE_NEEDED2	0x5250	SOL	4	056A	056A	056A
SO_PRIM_STORAGE_NEEDED3	0x5258	SOL	4	056E	056E	056E
SO_WRITE_OFFSET0	0x5280	SOL	2	0572	0572	0572
SO_WRITE_OFFSET1	0x5284	SOL	2	0574	0574	0574
SO_WRITE_OFFSET2	0x5288	SOL	2	0576	0576	0576
SO_WRITE_OFFSET3	0x528C	SOL	2	0578	0578	0578
3DSTATE_SO_BUFFER		SOL	32	057A	057A	057A
NOOP		SOL	1	059A	059A	059A
3DSTATE_SO_DECL_LIST		SOL	259	059B	059B	059B
NOOP		SOL	2	069E	069E	069E
3DSTATE_INDEX_BUFFER		VF	5	06A0	06A0	06A0
3DSTATE_VERTEX_BUFFERS		VF	133	06A5	06A5	06A5
3DSTATE_VERTEX_ELEMENTS		VF	69	072A	072A	072A
3DSTATE_VF_STATISTICS		VF	1	076F	076F	076F
3DSTATE_VF		VF	2	0770	0770	0770
3DSTATE_SGVS		VF	2	0772	0772	0772
3DSTATE_VF_INSTANCING		VF	69	0774	0774	0774

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
3DSTATE_VF_TOPOLOGY		VF	2	07B9	07B9	07B9
Load_Register_Immediate header	0x1100_1095	VF	1	07BB	07BB	07BB
INSTANCE CNT	08300 - 08384h	VF	68	07BC	07BC	07BC
INSTANCE INDX	08400 - 08484h	VF	68	0800	0800	0800
COMMITTED VERTEX NUMBER	08390h	VF	2	0844	0844	0844
COMMITTED INSTANCE ID	08394h	VF	2	0846	0846	0846
COMMITTED PRIMITIVE ID	08398h	VF	2	0848	0848	0848
STATUS	0839Ch	VF	2	084A	084A	084A
COMMON VERTEX	083A0h	VF	2	084C	084C	084C
VFSKPD—VF Scratch Pad (DEBUG)	083A8	VF	2	084E	084E	084E
				0852	0852	0852
				0854	0854	0854
				0856	0856	0856
NOOP		VF	14	0858	0858	0858
Load_Register_Immediate header	0x1100_105D	VFE	1	0860	0860	0860
TDL0 DATA		VFE	94	0861	0861	0861
NOOP		VFE	1	08BF	08BF	08BF
Load_Register_Immediate header	0x1100_105D	VFE	1	08C0	08C0	08C0
TDL1 DATA		VFE	94	08C1	08C1	08C1
NOOP		VFE	1	091F	091F	091F
Load_Register_Immediate header	0x1100_105D	VFE	1	0920	0920	0920
TDL2 DATA		VFE	94	0921	0921	0921
NOOP		VFE	1	097F	097F	097F
Load_Register_Immediate header	0x1100_105D	VFE	1	0980	0980	0980
TDL3 DATA		VFE	94	0981	0981	0981
NOOP		VFE	1	09DF	09DF	09DF
Load_Register_Immediate header	0x1100_105D	VFE	1	09E0	09E0	09E0
TDL4 DATA		VFE	94	09E1	09E1	09E1
NOOP		VFE	1	0A3F	0A3F	0A3F
Load_Register_Immediate header	0x1100_105D	VFE	1	0A40	0A40	0A40
TDL5 DATA		VFE	94	0A41	0A41	0A41
NOOP		VFE	1	0A9F	0A9F	0A9F
Load_Register_Immediate header	0x1100_105D	VFE	1	0AA0	0AA0	0AA0
TDL6 DATA		VFE	94	0AA1	0AA1	0AA1
NOOP		VFE	1	0AFF	0AFF	0AFF

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
Load_Register_Immediate header	0x1100_105D	VFE	1	0B00	0B00	0B00
TDL7 DATA		VFE	94	0B01	0B01	0B01
NOOP		VFE	1	0B5F	0B5F	0B5F
Load_Register_Immediate header	0x1100_105D	VFE	1	0B60	0B60	0B60
TDL8 DATA		VFE	94	0B61	0B61	0B61
NOOP		VFE	1	0BBF	0BBF	0BBF
Load_Register_Immediate header	0x1100_1023	VFE	1	0BC0	0BC0	0BC0
GW0 DATA		VFE	36	0BC1	0BC1	0BC1
NOOP		VFE	11	0BE5	0BE5	0BE5
Load_Register_Immediate header	0x1100_1023	VFE	1	0BF0	0BF0	0BF0
GW1 DATA		VFE	36	0BF1	0BF1	0BF1
NOOP		VFE	11	0C15	0C15	0C15
Load_Register_Immediate header	0x1100_1023	VFE	1	0C20	0C20	0C20
GW2 DATA		VFE	36	0C21	0C21	0C21
NOOP		VFE	11	0C45	0C45	0C45
Load_Register_Immediate header	0x1100_1023	VFE	1	0C50	0C50	0C50
GW3 DATA		VFE	36	0C51	0C51	0C51
NOOP		VFE	11	0C75	0C75	0C75
Load_Register_Immediate header	0x1100_1023	VFE	1	0C80	0C80	0C80
GW4 DATA		VFE	36	0C81	0C81	0C81
NOOP		VFE	11	0CA5	0CA5	0CA5
Load_Register_Immediate header	0x1100_1023	VFE	1	0CB0	0CB0	0CB0
GW5 DATA		VFE	36	0CB1	0CB1	0CB1
NOOP		VFE	11	0CD5	0CD5	0CD5
Load_Register_Immediate header	0x1100_1023	VFE	1	0CE0	0CE0	0CE0
GW6 DATA		VFE	36	0CE1	0CE1	0CE1
NOOP		VFE	11	0D05	0D05	0D05
Load_Register_Immediate header	0x1100_1023	VFE	1	0D10	0D10	0D10
GW7 DATA		VFE	36	0D11	0D11	0D11
NOOP		VFE	11	0D35	0D35	0D35
Load_Register_Immediate header	0x1100_1023	VFE	1	0D40	0D40	0D40
GW8 DATA		VFE	36	0D41	0D41	0D41
NOOP		VFE	11	0D65	0D65	0D65
Load_Register_Immediate header	0x1100_1017	VFE	1	0D70	0D70	0D70
TSG0 DATA		VFE	24	0D71	0D71	0D71

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
NOOP		VFE	7	0D89	0D89	0D89
Load_Register_Immediate header	0x1100_1017	VFE	1	0D90	0D90	0D90
TSG1 DATA		VFE	24	0D91	0D91	0D91
NOOP		VFE	7	0DA9	0DA9	0DA9
Load_Register_Immediate header	0x1100_1017	VFE	1	0DB0	0DB0	0DB0
TSG2 DATA		VFE	24	0DB1	0DB1	0DB1
NOOP		VFE	7	0DC9	0DC9	0DC9
Load_Register_Immediate header	0x1100_1017	VFE	1	0DD0	0DD0	0DD0
TSG3 DATA		VFE	24	0DD1	0DD1	0DD1
NOOP		VFE	7	0DE9	0DE9	0DE9
Load_Register_Immediate header	0x1100_1017	VFE	1	0DF0	0DF0	0DF0
TSG4 DATA		VFE	24	0DF1	0DF1	0DF1
NOOP		VFE	7	0E09	0E09	0E09
Load_Register_Immediate header	0x1100_1017	VFE	1	0E10	0E10	0E10
TSG5 DATA		VFE	24	0E11	0E11	0E11
NOOP		VFE	7	0E29	0E29	0E29
Load_Register_Immediate header	0x1100_1017	VFE	1	0E30	0E30	0E30
TSG6 DATA		VFE	24	0E31	0E31	0E31
NOOP		VFE	7	0E49	0E49	0E49
Load_Register_Immediate header	0x1100_1017	VFE	1	0E50	0E50	0E50
TSG7 DATA		VFE	24	0E51	0E51	0E51
NOOP		VFE	7	0E69	0E69	0E69
Load_Register_Immediate header	0x1100_1017	VFE	1	0E70	0E70	0E70
TSG8 DATA		VFE	24	0E71	0E71	0E71
NOOP		VFE	7	0E89	0E89	0E89
Load_Register_Immediate header	0x1100_1009	VFE	1	0E90	0E90	0E90
VFE Data		VFE	10	0E91	0E91	0E91
NOOP		VFE	5	0E9B	0E9B	0E9B
MEDIA_VFE_STATE		VFE	9	0EA0	0EA0	0EA0
MEDIA_CURBE_LOAD		VFE	4	0EA9	0EA9	0EA9
MEDIA_INTERFACE_DESCRIPTOR_LOAD		VFE	4	0EAD	0EAD	0EAD
NOOP		VFE	15	0EB1	0EB1	0EB1
NOOP		RS	2	0EC0	0EC0	0EC0
3DSTATE_BINDING_TABLE_POOL_ALLOC		RS	4	0EC2		0EC2
3DSTATE_BINDING_TABLE_EDIT_VS		RS	258	0EC6		0EC6

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
NOOP		RS	6	0FC8		0FC8
3DSTATE_BINDING_TABLE_EDIT_GS		RS	258	0FCE		0FCE
NOOP		RS	6	10D0		10D0
3DSTATE_BINDING_TABLE_EDIT_HS		RS	258	10D6		10D6
NOOP		RS	6	11D8		11D8
3DSTATE_BINDING_TABLE_EDIT_DS		RS	258	11DE		11DE
NOOP		RS	6	12E0		12E0
3DSTATE_BINDING_TABLE_EDIT_PS		RS	258	12E6		12E6
3DSTATE_GATHER_POOL_ALLOC		RS	4	13E8		13E8
MI_BATCH_BUFFER_END/NOOP ***		RS	1	13EC		13EC
NOOP		RS	5	13ED		13ED
3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC		RS	4	13F2		13F2
3DSTATE_DX9_CONSTANTF_VS(Global)		RS	1026	13F6		13F6
NOOP		RS	6	17F8		17F8
3DSTATE_DX9_CONSTANTI_VS(Global)		RS	130	17FE		17FE
NOOP		RS	6	1880		1880
3DSTATE_DX9_CONSTANTB_VS(Global)		RS	18	1886		1886
NOOP		RS	6	1898		1898
3DSTATE_DX9_CONSTANTF_VS(local)		RS	1026	189E		189E
NOOP		RS	6	1CA0		1CA0
3DSTATE_DX9_CONSTANTI_VS(local)		RS	130	1CA6		1CA6
NOOP		RS	6	1D28		1D28
3DSTATE_DX9_CONSTANTB_VS(local)		RS	18	1D2E		1D2E
NOOP		RS	3	1D40		1D40
3DSTATE_DX9_LOCAL_VALID_VS		RS	11	1D43		1D43
3DSTATE_DX9_CONSTANTF_PS(Global)		RS	1026	1D4E		1D4E
NOOP		RS	6	2150		2150
3DSTATE_DX9_CONSTANTI_PS(Global)		RS	130	2156		2156
NOOP		RS	6	21D8		21D8
3DSTATE_DX9_CONSTANTB_PS(Global)		RS	18	21DE		21DE
NOOP		RS	6	21F0		21F0
3DSTATE_DX9_CONSTANTF_PS(local)		RS	1026	21F6		21F6
NOOP		RS	6	25F8		25F8
3DSTATE_DX9_CONSTANTI_PS(local)		RS	130	25FE		25FE

Description	MMIO Offset/ Command	Unit	# of DW	Address Offset (PWR)	Address Offset (No Ext.)	Address Offset (Ext.)
NOOP		RS	6	2680		2680
3DSTATE_DX9_CONSTANTB_PS(local)		RS	18	2686		2686
3DSTATE_DX9_LOCAL_VALID_PS		RS	11	2698		2698
MI_BATCH_BUFFER_END		RS	1	26A3		26A3
NOOP		RS	12	26A4		26A4
URB_ATOMIC_STORAGE		GAFS	8192	26B0		26B0
				46B0	0EC0	46B0
			DW	18096	3776	18096
			Kbytes	70.6875	14.75	70.6875

Ring Buffer

Ring Buffer can exist anywhere in memory mapped via Global GTT. Ring buffer details are mentioned in the ring context area of LRCA (Ring Buffer - Start Address, Head Offset, Tail Pointer & Control Register) in Execution List mode of scheduling. Ring Buffer registers are directly programmed in Ring Buffer mode of scheduling.

Ring Context

Ring Context starts at 4K offset from LRCA. Ring context contains all the details that are needed to be initialized by SW for submitting a context to HW for execution (Ring Buffer Details, Page Directory Information, and so on). Ring context is five cachelines in size. Note that the last cacheline of the ring context is specific for a given Engine and hence SW needs to populate it accordingly.

Ring Context comprises of the EXECLIST CONTEXT, EXECLIST CONTEXT (PPGTT Base) of the register state context. In Ring Buffer mode of scheduling EXECLIST CONTEXT contents are save/restored as NOOPS by HW.

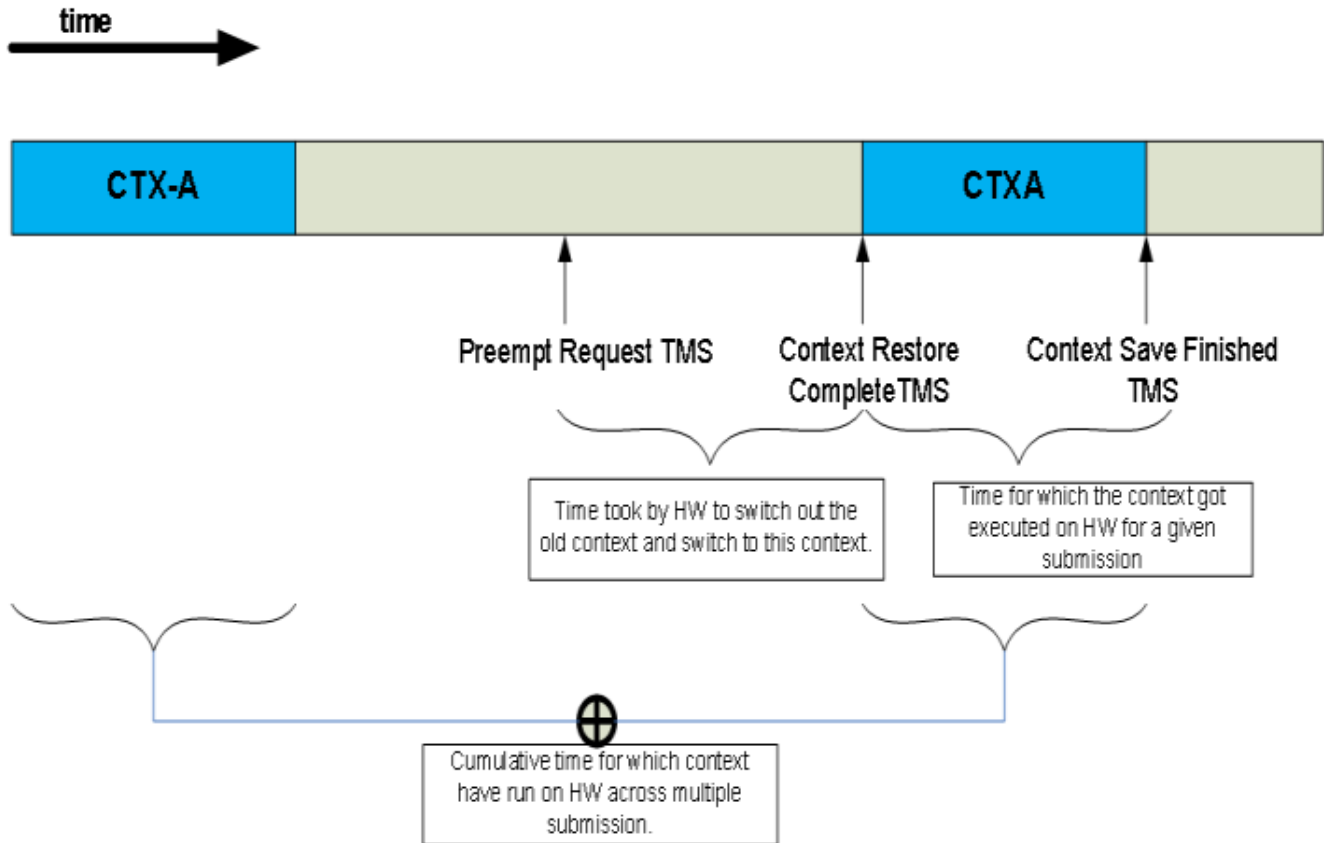
EXECLIST CONTEXT
EXECLIST CONTEXT(PPGTT Base)

The Per-Process Hardware Status Page

The following table defines the layout of the Per-Process Hardware Status Page.

The DWord offset values in the table are in hexadecimal.

DWord Offset	Description	
(3FFh – 020h)	These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions.	
1F:1C	Reserved.	
1B:18	MI_SEMAPHORE_WAIT. MI_SEMAPHORE_WAIT command on which the context got switched out due to semaphore wait. This field is only valid and must be looked at when the context switch reason in context status buffer is stated as "Wait on Semaphore".	
17:16	Context Save Finished Timestamp. TIMESTAMP register sampled on context save completion is reported.	
15:14	Context Restore Complete Timestamp. TIMESTAMP register sampled on context restore complete is reported.	
13:12	Preempt Request Received Timestamp. TIMESTAMP register sampled on preemption request is reported.	
11	Bits	Description
	31:2	Reserved Format: MBZ Must be set to modify corresponding bit in Bits 15:0. (All implemented bits)
	1	Reserved Format: MBZ Must be set to modify corresponding bit in Bits 15:0. (All implemented bits)
10	Cumulative Context Run Time. This has the cumulative run time of the context on HW. HW reports CTX_TIMESTAMP to this location on a context switch.	
F:5	Reserved.	
4	Ring Head Pointer Storage. The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an "automatic report" (see RINGBUF registers).	
3:0	Reserved.	



This page is designed to be read by SW to glean additional details about a context beyond what it can get from the context status.

Accesses to this page are automatically treated as cacheable and snoopable. It is therefore illegal to locate this page in any region where snooping is illegal (such as in stolen memory).

Render Engine Power Context

Table below captures the data from CS power context save/restored by PM. Address offset in the below table is relative to the starting location of CS in the overall power context image managed by PM.

Description	Offset	Unit	# of DW	Address Offset (PWR)
NOOP		CS	1	0
Load_Register_Immediate header	0x1100_10A7	CS	1	0001
GFX_MODE	0x229C	CS	2	0002
GHWSP	0x2080	CS	2	0004
RING_BUFFER_CONTROL (Ring Always Disabled)	0x203C	CS	2	0006
Ring Head Pointer Register	0x2034	CS	2	0008
Ring Tail Pointer Register	0x2030	CS	2	000A
RING_BUFFER_START	0x2038	CS	2	000C
RING_BUFFER_CONTROL (Original status)	0x203C	CS	2	000E
Batch Buffer Current Head Register (UDW)	0x2168	CS	2	0010
Batch Buffer Current Head Register	0x2140	CS	2	0012
Batch Buffer State Register	0x2110	CS	2	0014
SECOND_BB_ADDR_UDW	0x211C	CS	2	0016
SECOND_BB_ADDR	0x2114	CS	2	0018
SECOND_BB_STATE	0x2118	CS	2	001A
RC_PSMI_CONTROL	0x2050	CS	2	001C
RC_PWRCTX_MAXCNT	0x2054	CS	2	001E
CTX_WA_PTR	0x2058	CS	2	0020
NOPID	0x2094	CS	2	0026
HWSTAM	0x2098	CS	2	0028
FF_THREAD_MODE	0x20A0	CS	2	002A
IMR	0x20A8	CS	2	002C
EIR (Restoring EIR doesn't it clear)	0x20B0	CS	2	002E
EMR	0x20B4	CS	2	0030
CMD_CCTL_0	0x20C4	CS	2	0032
GAFS_Mode	0x212C	CS	2	0038
UHPTR	0x2134	CS	2	003A
BB_PREEMPT_ADDR_UDW	0x216C	CS	2	003C
BB_PREEMPT_ADDR	0x2148	CS	2	003E
RING_BUFFER_HEAD_PREEMPT_REG	0x214C	CS	2	0040
CXT_SIZE	0x21A8	CS	2	0042
CXT_OFFSET	0x21AC	CS	2	0044
PREEMPT_DLY	0x2214	CS	2	0046

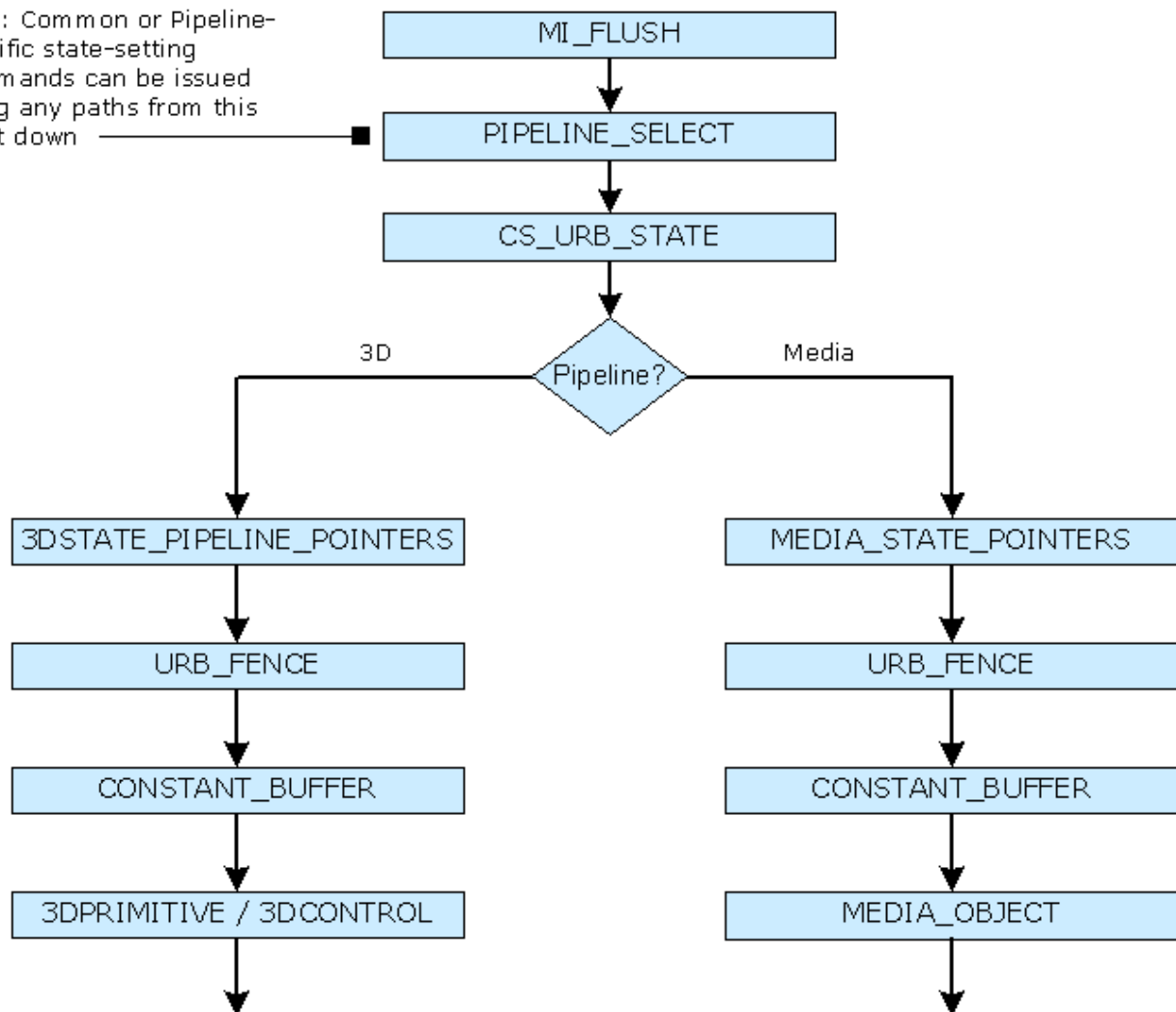
Description	Offset	Unit	# of DW	Address Offset (PWR)
MTCH_CID_RST (should not be required any more)	0x222C	CS	2	0048
SYNC_FLIP_STATUS	0x22D0	CS	2	004C
SYNC_FLIP_STATUS_1	0x22D4	CS	2	004E
WAIT_FOR_RC6_EXIT	0x20CC	CS	2	0054
RS_OFFSET	0x21B4	CS	2	0056
RCS_CTXID_PREEMPTION_HINT	0x24CC	CS	2	0058
RS_PREEMPTION_HINT_UDW	0x24C4	CS	2	005A
RS_PREEMPTION_HINT	0x24C0	CS	2	005C
CS_PREEMPTION_HINT_UDW	0x24C8	CS	2	005E
CS_PREEMPTION_HINT	0x24BC	CS	2	0060
CCID Register	0x2180	CS	2	0062
SBB_PREEMPT_ADDRESS_UDW	0x2138	CS	2	0064
SBB_PREEMPT_ADDRESS	0x213C	CS	2	0066
URB_CTX_OFFSET	0x21B8	CS	2	0068
MI_PREDICATE_RESULT_2	0x23BC	CS	2	006A
CTXT_ST_PTR	0x23A0	CS	2	006C
CTXT_ST_BUF	0x2370	CS	24	006E
SEMA_WAIT_POLL	0x224C	CS	2	0086
IDLEDELAY	0x223C	CS	2	0088
DISPLAY MESSAGE FORWARD STATUS	0x22E8	CS	2	008A
RCS_FORCE_TO_NONPRIV	0x24D0	CS	24	008C
VF PREEMPTION VERTEX HINT	0x83B0	VF	2	00A4
VF PREEMPTION INSTANCE HINT	0x83B4	VF	2	00A6
EXECLIST_STATUS_REGISTER	0x2234	CS	2	00A8
NOOP		CS	20	00AA
NOOP		CS	1	00BE
MI_BATCH_BUFFER_END		CS	1	00BF

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.

The following flowchart illustrates an example ordering of commands which can be used to perform activity within the GPE.

Note: Common or Pipeline-specific state-setting commands can be issued along any paths from this point down



B6680-01

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.

RS_PREEMPT_STATUS_UDW - 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

Programming Note	
Context:	Resource Streamer Sync Commands.
If resource streamer is enabled in a batch buffer, an MI_RS_STORE_DATA_IMM with Resource Streamer Flush set must be programmed before any Resource Streamer Sync Command.	

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.

Common Abbreviations

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 hardware.
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

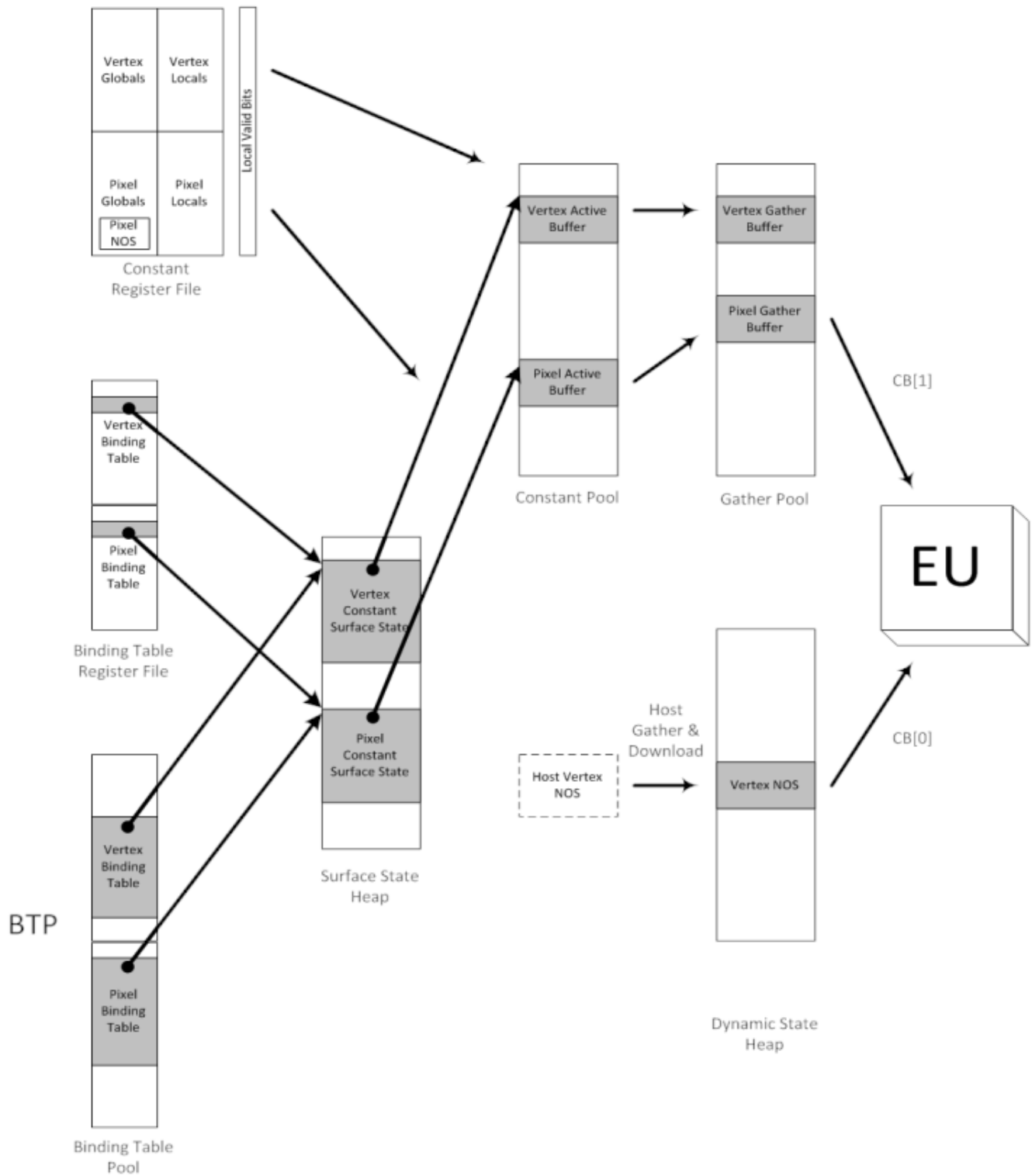
This section briefly describes the operation of the Resource Streamer. Specifically, it calls out reset state, initialization requirements, and major operational tasks of the RS.

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.

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 be 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 saved for global and 1 for local register per FF. There will be 1 3DSTATE_CONSTANTB_* will be saved for global and 1 for local register per FF.

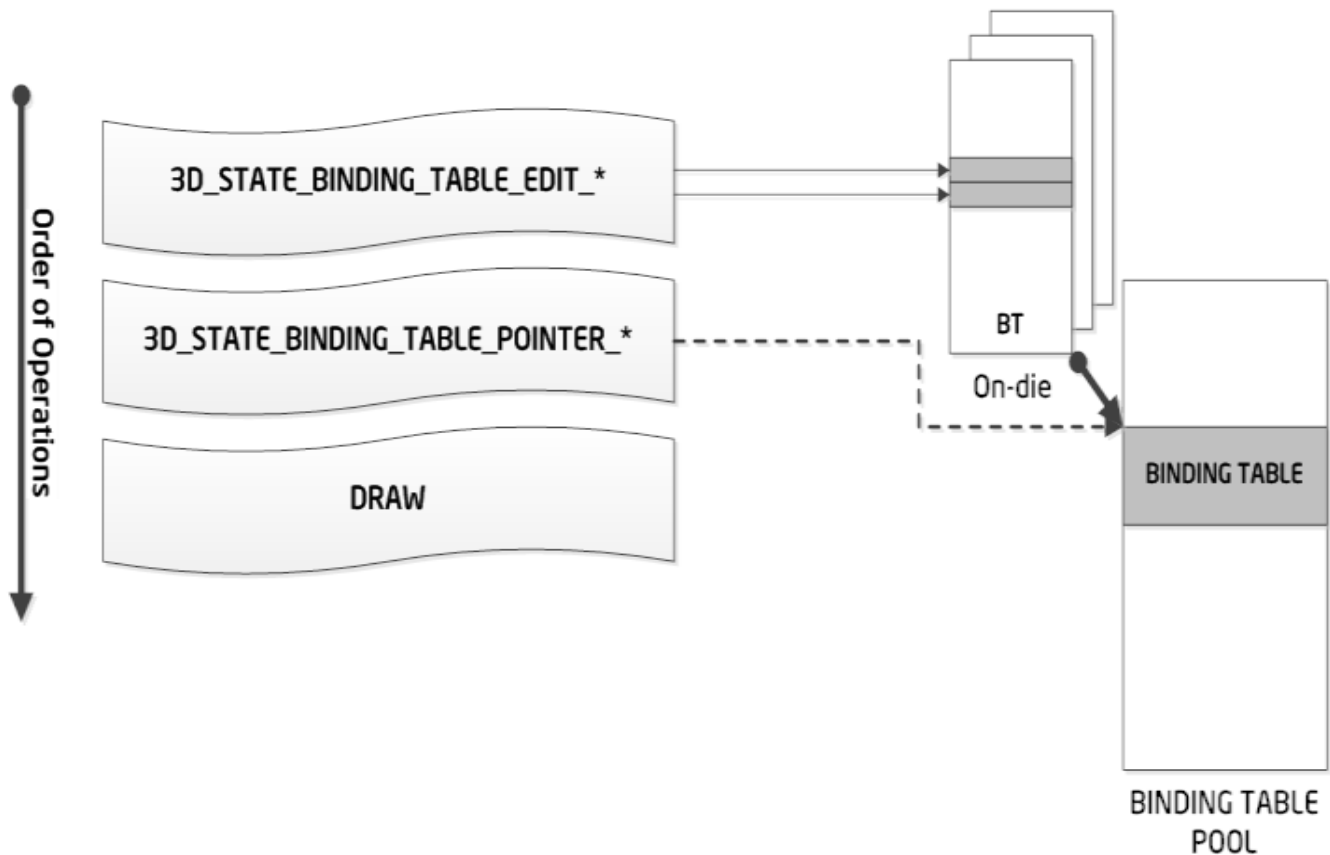
Gather Push Constants Image

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_PS	1026
3DSTATE_CONSTANTI_PS	130
3DSTATE_CONSTANTB_PS	18
3DSTATE_LOCAL_VALID_VS	10
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
0	1	Illegal (Undefined)
1	0	Illegal **
1	1	HW Generate BT
Table Notes * Active means with RS enabled in Batch buffer and MI_RS_CONTROL field with RS on. ** If RS is enabled, Binding table pool is required to be enabled.		

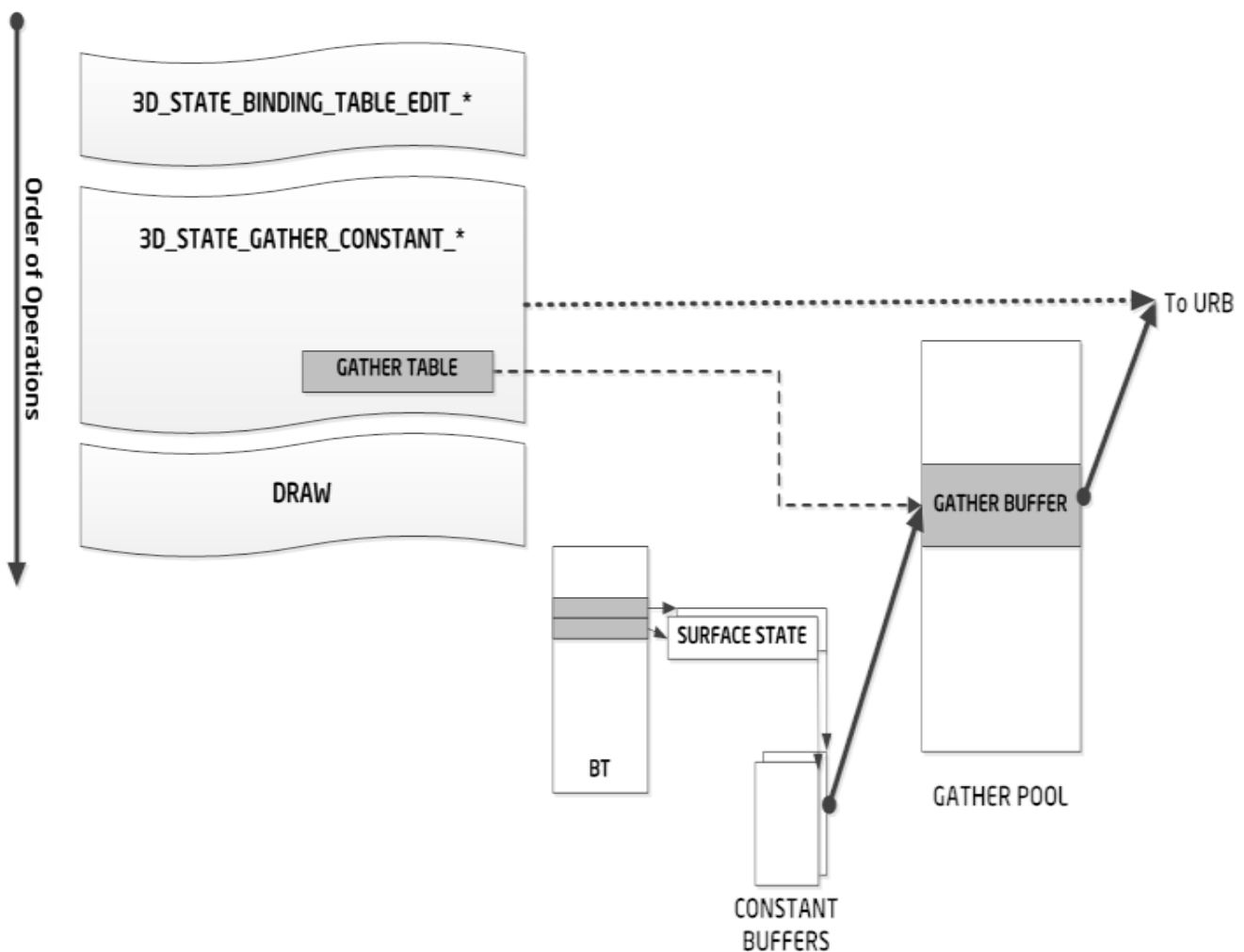
Gather Push Constants

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 a 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

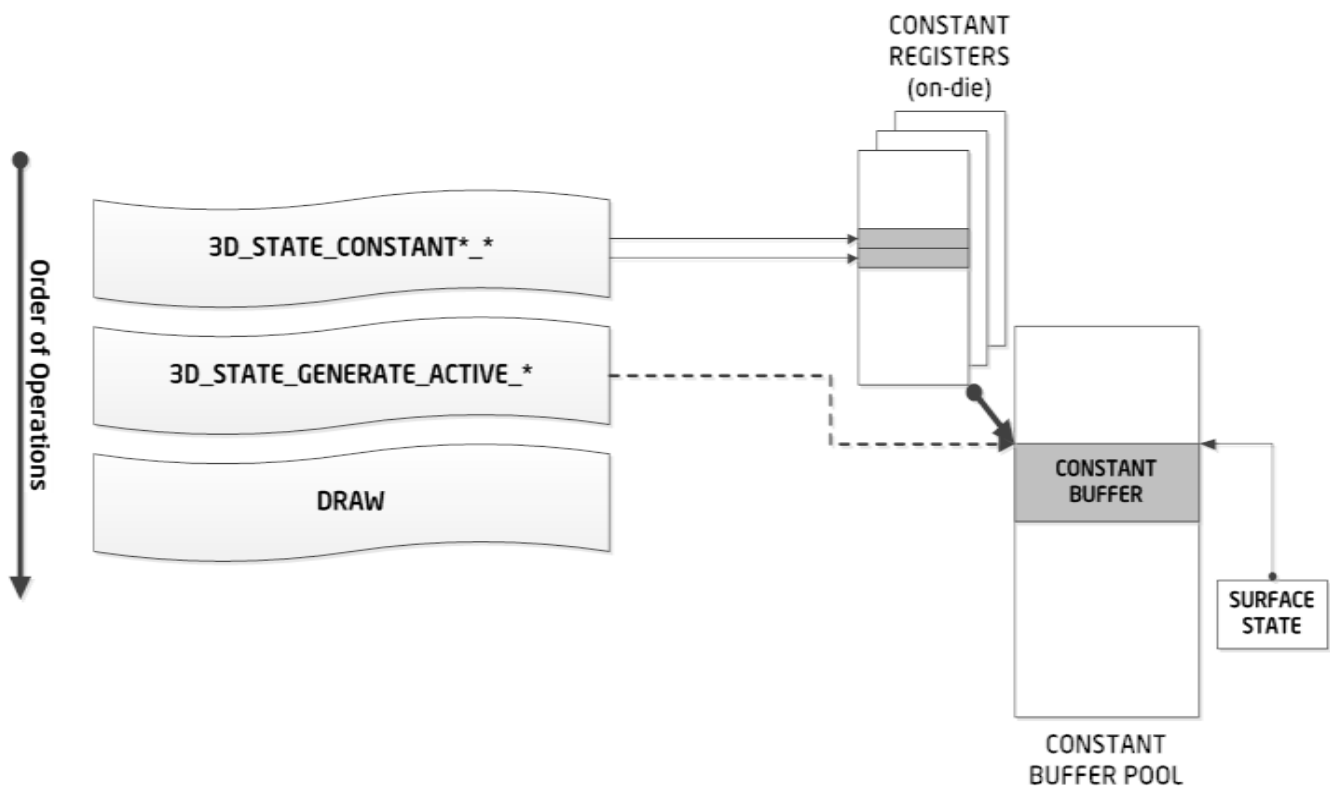


Constant Buffer Generation (not DX9)

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 Handling (No Perf)	RS Handling (Perf)
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			

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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_POINTER S			Refer to 3D Pipeline
3h	0h	10h	3DSTATE_VS			Refer to Vertex Shader
3h	0h	11h	3DSTATE_GS			Refer to Geometry Shader
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

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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 Tessellator
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_POINTER S_SF_CLIP			Refer to Strips & Fans
3h	0h	23h	3DSTATE_VIEWPORT_STATE_POINTER S_CC			Refer to Windower
3h	0h	24h	3DSTATE_BLEND_STATE_POINTERS			Refer to Pixel Shader
3h	0h	25h	3DSTATE_DEPTH_STENCIL_STATE_POI NTERS			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	

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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	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	
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

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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	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			

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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			
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	

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command	RS Handling (No Perf)	RS Handling (Perf)	Notes
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.
If the Binding Table Pool Enable is cleared while the Resource Streamer is enabled within a batch buffer, then the on chip storage for the binding table will not be context save and restored. To save the Binding Table Pool, before disabling the Pool enable, disable the resource streamer thru the MI_RS_CONTROL command. And then, before reenabling the Binding Table Pool, reenable the resource streamer thru the MI_RS_CONTROL command.

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

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.	

Programming Note	
Context:	Dx9 Constant Buffer generation.
<ul style="list-style-type: none"> 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

DX9 Constant Buffer Map

Offset		Cache Line
0x0000	256 4-component Float Constants	0
0x0fff		63
0x1000	32 4-component integer constants	64
0x107f		
0x1080	16 1-component boolean constants	72
0x108f		
0x1090	unused	
0x1fff		

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

DX9 Constant Buffer Map

Offset		Cache Line
0x0000	256 4-component Float Constants	0
0x0fff		63
0x1000	32 4-component integer constants	64
0x107f		
0x1080	16 1-component boolean constants	72
0x108f		
0x1090	unused	
0x1fff		

3DSTATE_DX9_GENERATE_ACTIVE_PS

Shared Functions

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.

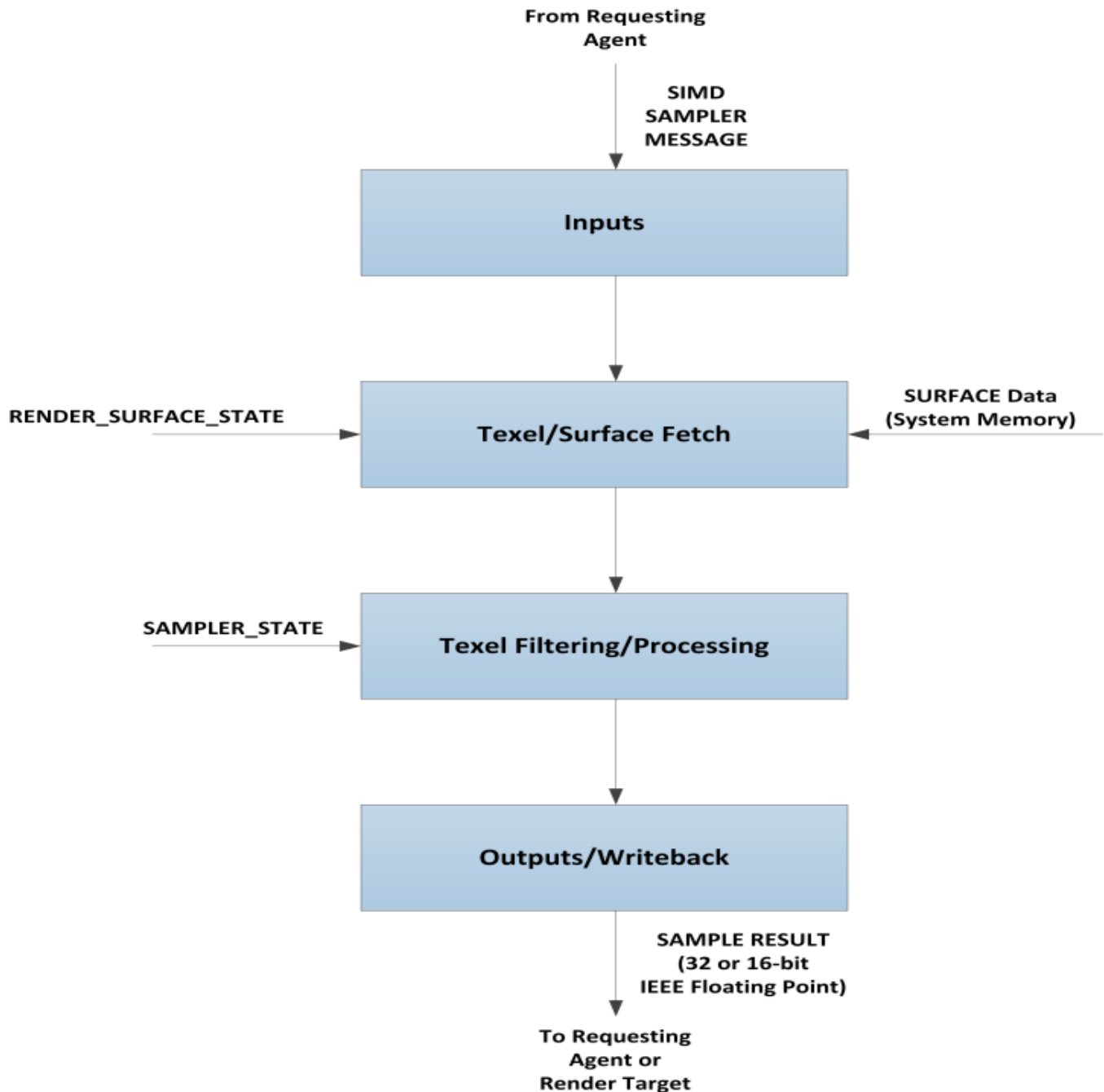
The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the Gen Core to complete the *sample* instruction.

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 Generation	The Sampling Engine determines the required set of texel samples (specific texel values from specific texture maps), as defined by the texture map parameters and filtering modes. This includes coordinate wrap/clamp/mirror control, mipmap LOD computation and sample and/or miplevel weighting factors to be used in the subsequent filtering operations.
Texel Fetch	The required texel samples are read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.
Texture Palette Lookup	For streams which have "paletted" texture surface formats, this function uses the "index" values read from the texture map to look up texel color data from the texture palette.
Shadow Pre-Filter Compare	For shadow mapping, the texel samples are first compared to the 3rd (R) component of the pixel's texture coordinate. The boolean results are used in the texture filter.
Texel Filtering	Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This "combination" ranges from simply passing through a "nearest" sample to blending the results of anisotropic filters performed on two mipmap levels. The output of this function is a single 4-component texel value.
Texel Color Gamma Linearization	Performs optional gamma decorrection on texel RGB (not A) values.
8x8 Video Scaler	Performs scaling using an 8x8 filter.

Sampling Engine

3D Sampler Theory of Operation

The 3D sampler (sometimes referred to as texture sampler) is a self-contained functional block within the 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 kernel 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.

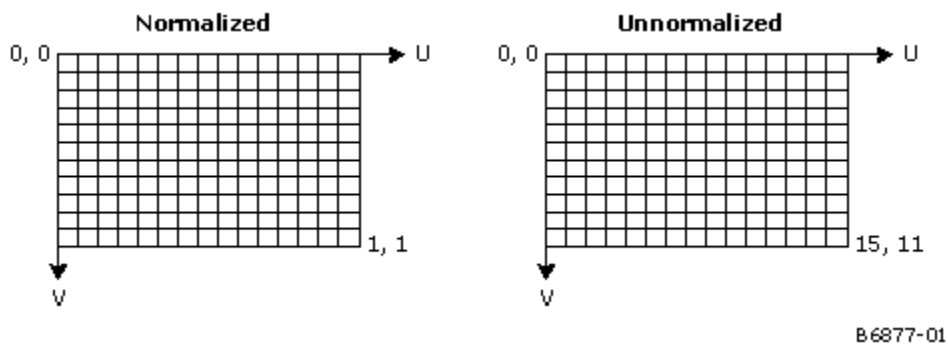
Texture Coordinate Normalization

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 located at the upper/left edge of the upper left texel of the base texture map.

Normalized vs. Unnormalized Texture Coordinates



B 6877-01

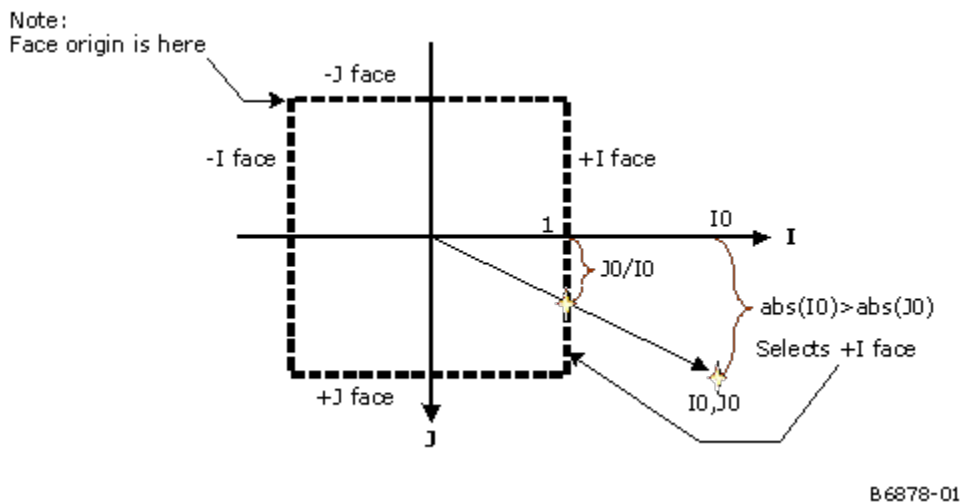
Texture Coordinate Computation

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

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

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

Cube Map Coordinate Computation Example



Texel Address Generation

To better understand texture mapping, consider the mapping of each object (screen-space) pixel onto the texture 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 $\frac{1}{2}$ the resolution of the previous level in the U and V directions (to a minimum of 1 texel in either direction) until a 1x1 texture map is reached. The dimensions of mipmap levels need not be a power of 2.

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

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

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

Base Level Of Detail (LOD)

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

The ideal LOD computation is included below.

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

where :

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

LOD Bias

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

The LOD bias is defined as sum of the *LODBias* state variable and the *pixLODBias* input from the input message (which can be non-zero only for sample_b messages). The application of LOD Bias is unconditional, therefore these variables must both be set to zero in order to prevent any undesired biasing.

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

LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable. 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 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 $\text{LOD} = 0$ 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) mip level 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 mip levels.

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

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

MagMode = (LOD - Base <= 0)

MagClampMipNone = LODClampMagnificationMode == MAG_CLAMP_MIPNONE

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

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

    LOD += 0.5
    LOD = floor(LOD)
else
    // MipFlt = Linear
    LOD = min(LOD, AdjMaxLod)
  
```

```

LOD = max(LOD, AdjMinLod)
TriBeta = frac(LOD)
LOD0 = floor(LOD)
LOD1 = LOD0 + 1

```

```

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

```

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

Intra-Level Filtering Setup

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

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

The following table summarizes the intra-level filtering modes.

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

MAPFILTER_NEAREST

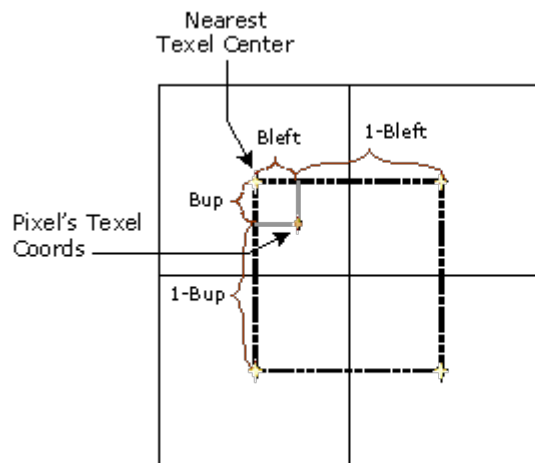
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level. This is a form of "Point Sampling".

MAPFILTER_LINEAR

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

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

Bilinear Filter Sampling

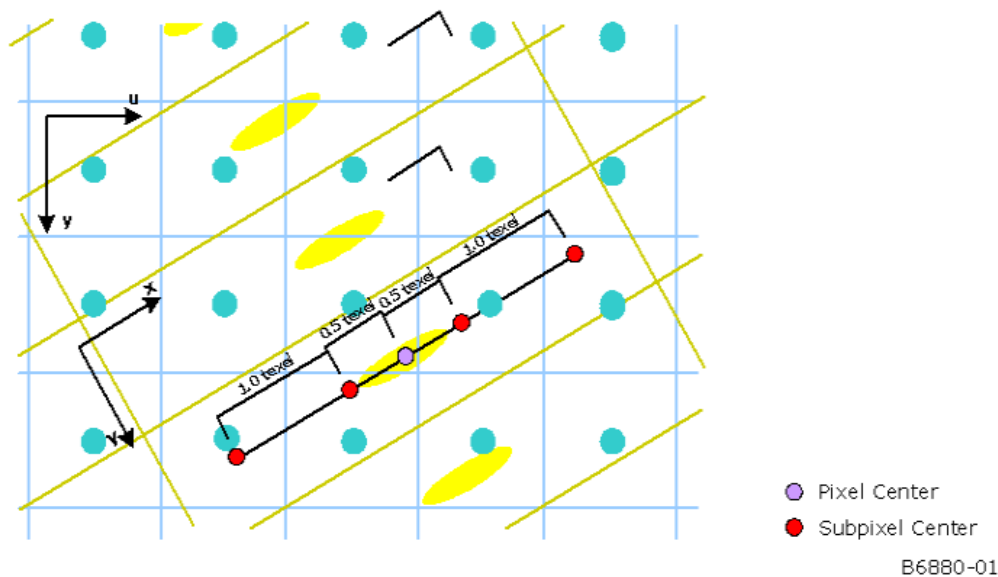


B.6879-01

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).

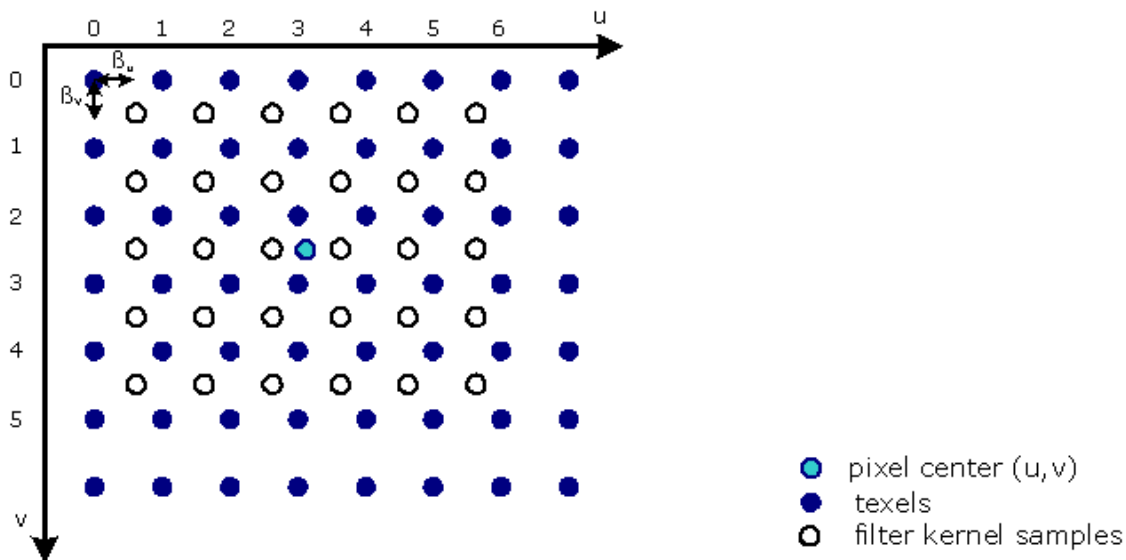
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f, v_f) relative to the pixel center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



B6881-01

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:

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

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

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

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

Texture Address Control

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

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

<i>TC[X,Y,Z] Control</i>	<i>Operation</i>
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 <code>SAMPLER_STATE</code> .
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.
TEXCOORDMODE_MIRROR_ONCE	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.

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

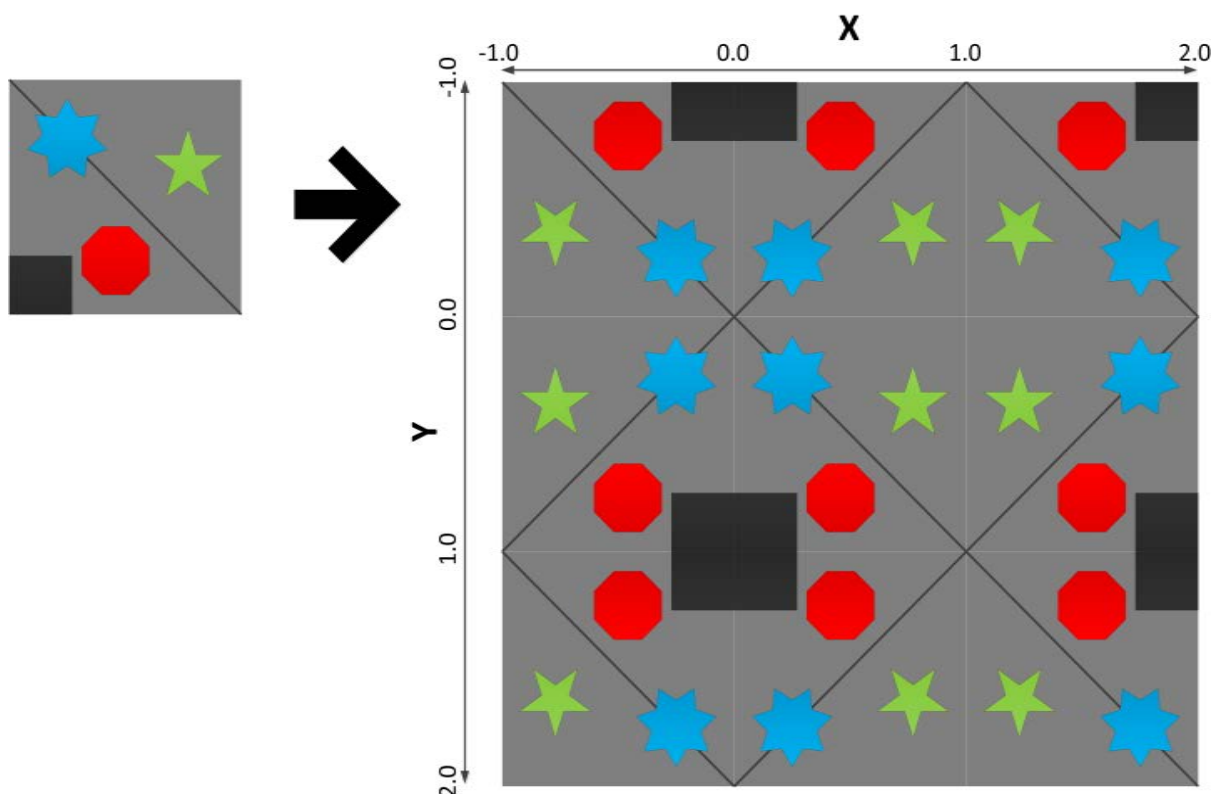
The number of map wraps on a given object is limited to 32. Going beyond this limit is legal, but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the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

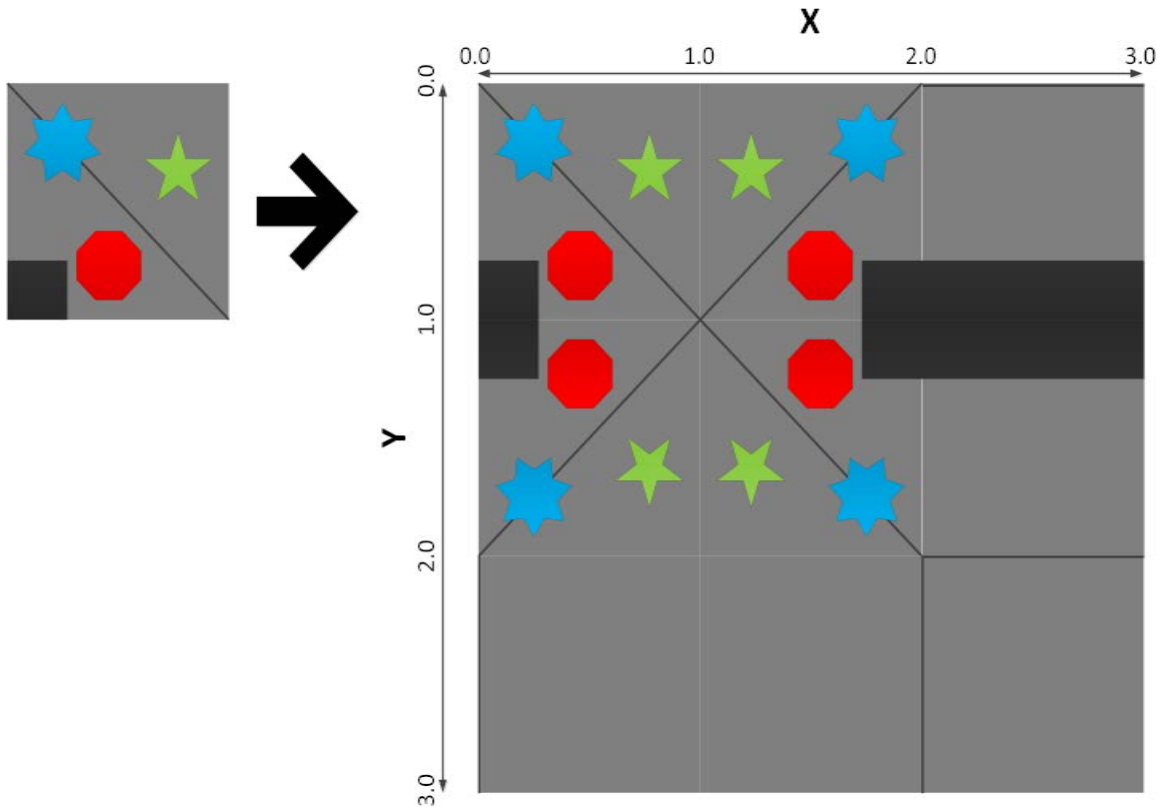
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR for both TCX and TCY is mapped.



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.

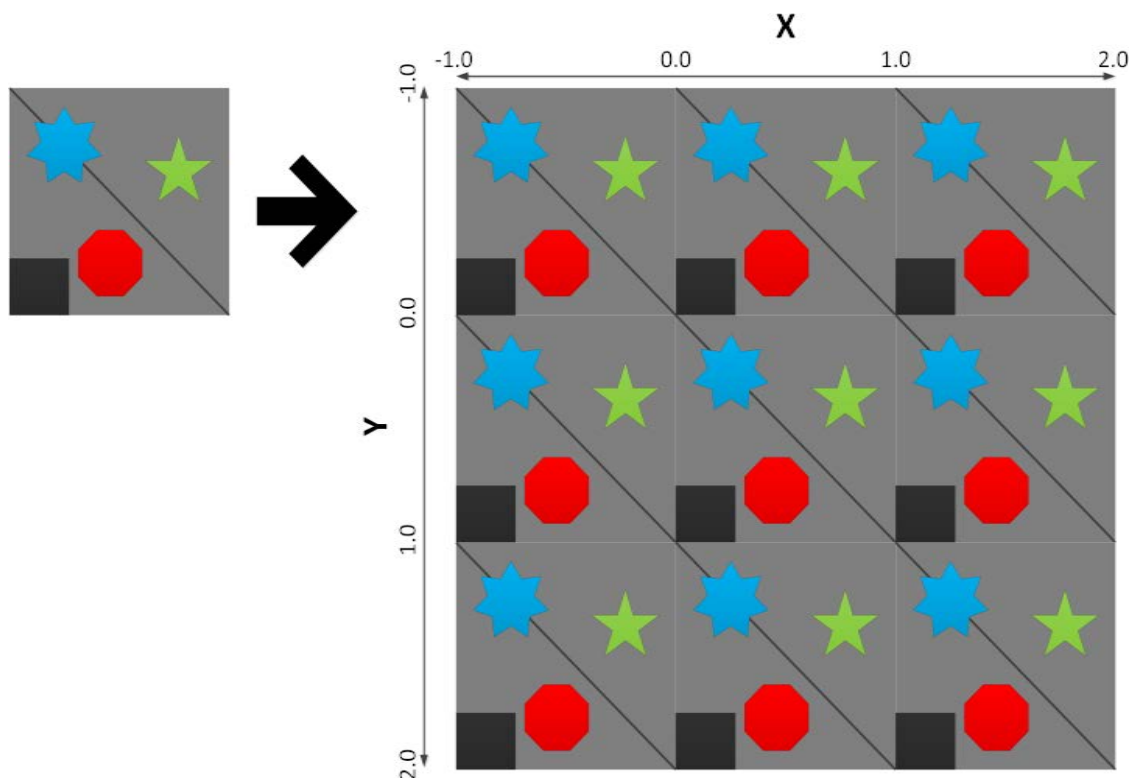
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR_ONCE for both TCX and TCY is mapped.



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.

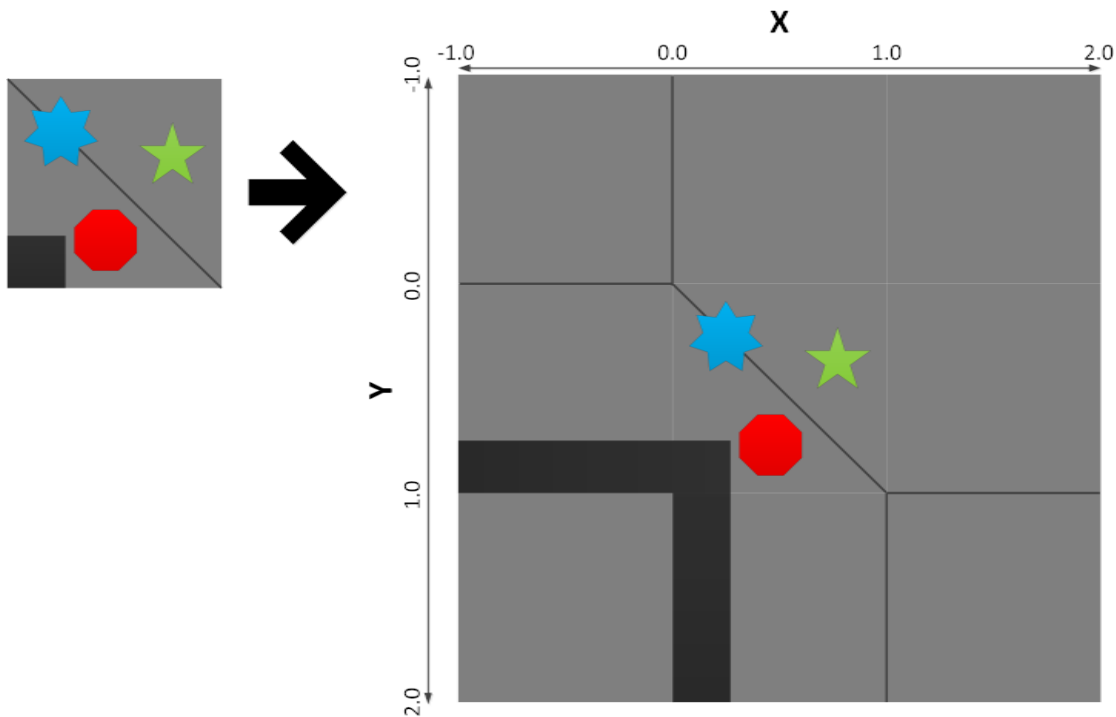


TEXCOORDMODE_CLAMP Mode

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.

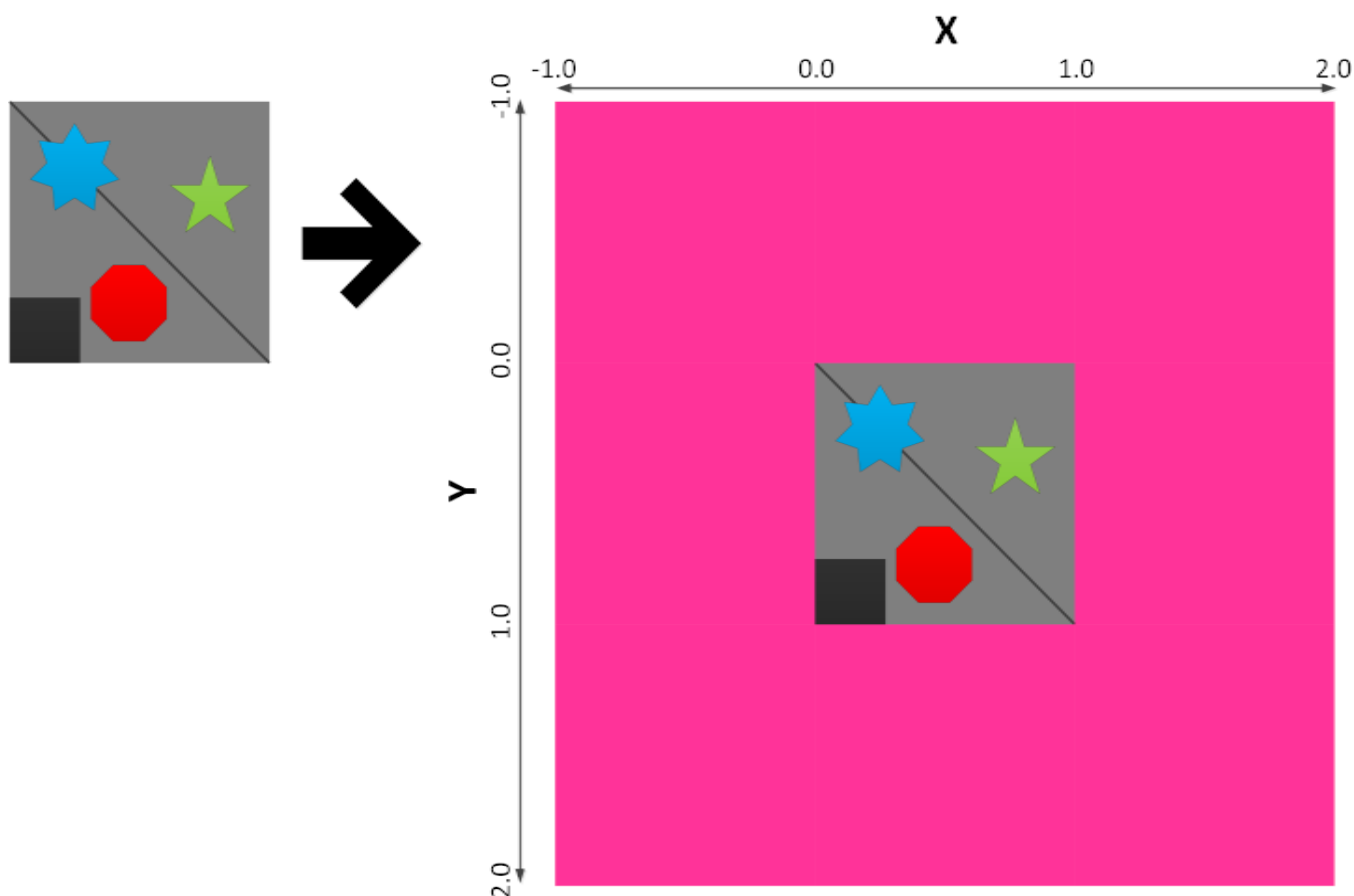
Texture Clamp Mode



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

Texel Chroma Keying

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

Chroma Key Testing

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

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

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

Chroma Key Effects

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

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

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

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

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

Shadow Prefilter Compare

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

<i>ShadowFunction</i>	<i>Result</i>
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

The *ld* message has added an additional parameter for sample index (*si*) to support unfiltered loading from a multisampled surface.

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

Any of the other messages (*sample**, *LOD*, *load4*) used with a (4x) multisampled surface would sample a surface with double the height and width as indicated in the surface state. Each pixel position on the original-sized surface is replaced with 2x2 samples that have the following arrangement:

sample 0	sample 2
sample 1	sample 3

This behavior is useful when implementing the multisample resolve operation by selecting *MAPFILTER_LINEAR* and rendering a full-screen rectangle half the size in each dimension of the source texture map (multisampled surface). If pixel offsets are set correctly, each pixel is the average of the four underlying samples.

Multisample Control Surface

Three new messages have been defined for the sampling engine, *ld_mcs*, *ld2dms*, 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 and *R32_UINT* for 8x surfaces, thus data is returned in unsigned integer format. Following the *ld_mcs*, the kernel issues a *ld2dms* message to sample the surface itself. The integer value from the MCS surface is delivered in the *mcs* parameter of this messages.

Since *sample* is no longer supported on multisampled surfaces, the multisample resolve must be done using *ld2dms*. For surfaces with **Multisampled Surface Storage Format** set to *MSFMT_MSS* and **MCS Enable** set to enabled, an optimization is available to enable higher performance for compressed pixels. The *ld2dss* message can be used to sample from a particular sample slice on the surface. By examining the MCS value, software can determine which sample slices to sample from. A simple optimization with probable large return in performance is to compare the MCS value to zero (indicating all samples are on sample slice 0), and sample only from sample slice 0 using *ld2dss* if MCS is zero. Sample slice 0 is the pixel color in this case. If MCS is not zero, each sample is then obtained using *ld2dms* messages and the results are averaged in the kernel after being returned. Refer to the multisample storage format in the GPU Overview volume for more details.

State

BINDING_TABLE_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.

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.

Sampler State Fetch

SAMPLER_STATE objects are fetched independently of surface state and cached locally in the 3D sampler independently (there may one or more SAMPLER_STATE objects associates with one or more RENDER_SURFACE_STATE objects). The sampler state is fetched using the **Sampler State Pointer (SSP)** which is provided either in the message header or directly from the command stream (message headers are not required). The **SSP** is an offset relative to the **Dynamic_State_Base_Address** and selects a table of 16 sampler states. The 4-bit **Sampler Index (SI)** in the message descriptor is used to select the specific **SAMPLER_STATE** object to be fetched from system memory and cached locally in the 3D sampler.

State Caching

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

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

SURFACE_STATE

The surface state is stored as individual elements, each with its own pointer in the binding table or its own entry in a memory heap in memory. Each surface state element is aligned to a 32-byte boundary.

Surface state defines the state needed for the following objects:

- texture maps (1D, 2D, 3D, cube) read by the sampling engine
- buffers read by the sampling engine
- constant buffers read by the data cache via the data port
- render targets read/written by the render cache via the data port
- streamed vertex buffer output written by the render cache via the data port
- media surfaces read from the texture cache or render cache via the data port
- media surfaces written to the render cache via the data port

The surface state definition can be found in the following section:

RENDER_SURFACE_STATE

Surface Formats

The RENDER_SURFACE_STATE contains a 9-bit field called **Surface Format**, which defines the exact format of the surface being sampled. The definition of the encodings for each supported format, including compressed formats can be found in the following section:

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.

Some formats are supported only in DX10/OGL **Border Color Mode**. Those formats have only that mode indicated. Formats that behave the same way in both **Border Color Modes** are indicated by that column being blank.

Surface Format Name	Filtering	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_UNORM					R	G	B	A					
R16G16B16A16_SNORM					R	G	B	A					
R16G16B16A16_SINT				DX10/OGL	R	G	B	A					
R16G16B16A16_UINT				DX10/OGL	R	G	B	A					
R16G16B16A16_FLOAT					R	G	B	A					
R32G32_FLOAT				DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R32G32_SINT				DX10/OGL	R	G	0.0	1.0	OCL				
R32G32_UINT				DX10/OGL	R	G	0.0	1.0	OCL				
R32_FLOAT_X8X24_TYPELESS		Yes		DX10/OGL	R	0.0	0.0	1.0	OCL				
X32_TYPELESS_G8X24_UINT				DX10/OGL	0.0	G	0.0	1.0	OCL				
L32A32_FLOAT				DX10/OGL	L	L	L	A					
R16G16B16X16_UNORM					R	G	B	1.0					
R16G16B16X16_FLOAT					R	G	B	1.0					
A32X32_FLOAT					0.0	0.0	0.0	A					
L32X32_FLOAT					L	L	L	1.0					
I32X32_FLOAT					I	I	I	I					
B8G8R8A8_UNORM			Yes		R	G	B	A					
B8G8R8A8_UNORM_SRGB					R	G	B	A					
R10G10B10A2_UNORM					R	G	B	A					
R10G10B10A2_UNORM_SRGB					R	G	B	A					
R10G10B10A2_UINT				DX10/OGL	R	G	B	A					

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	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					
R8G8B8A8_SINT				DX10/OGL	R	G	B	A					
R8G8B8A8_UINT				DX10/OGL	R	G	B	A					
R16G16_UNORM				DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R16G16_SNORM				DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R16G16_SINT				DX10/OGL	R	G	0.0	1.0	OCL				
R16G16_UINT				DX10/OGL	R	G	0.0	1.0	OCL				
R16G16_FLOAT				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
B10G10R10A2_UNORM					R	G	B	A					
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	OCL				
R32_UINT				DX10/OGL	R	0.0	0.0	1.0	OCL				
R32_FLOAT		Yes		DX10/OGL	R	0.0	0.0	1.0	OCL	R	1.0	1.0	1.0
R24_UNORM_X8_TYPELESS		Yes		DX10/OGL	R	0.0	0.0	1.0	OCL				
X24_TYPELESS_G8_UINT				DX10/OGL	0.0	G	0.0	1.0					
L16A16_UNORM					L	L	L	A					
I24X8_UNORM		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					

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
B4G4R4A4_UNORM			Yes		R	G	B	A					
B4G4R4A4_UNORM_SRGB					R	G	B	A					
R8G8_UNORM				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SNORM			Yes	DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SINT				DX10/OGL	R	G	0.0	1.0					
R8G8_UINT				DX10/OGL	R	G	0.0	1.0					
R16_UNORM		Yes		DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_SNORM				DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_SINT				DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_UINT				DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_FLOAT				DX10/OGL	R	0.0	0.0	1.0	OCL	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					
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					
P4A4_UNORM_PALETTE0					R	G	B	A					
A4P4_UNORM_PALETTE0					R	G	B	A					
P8_UNORM_PALETTE0					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					

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	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	OCL				
BC5_UNORM				DX10/OGL	R	G	0.0	1.0	OCL				
BC1_UNORM_SRGB					R	G	B	A					
BC2_UNORM_SRGB					R	G	B	A					
BC3_UNORM_SRGB					R	G	B	A					
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	BDW				R	G	B	1.0					
R8G8B8_SNORM	BDW				R	G	B	1.0					
BC4_SNORM				DX10/OGL	R	0.0	0.0	1.0	OCL				
BC5_SNORM				DX10/OGL	R	G	0.0	1.0	OCL				
R16G16B16_FLOAT					R	G	B	1.0					
R16G16B16_UNORM	BDW				R	G	B	1.0					
R16G16B16_SNORM	BDW				R	G	B	1.0					
BC6H_SF16	BDW				R	G	B	1.0*					
BC7_UNORM	BDW				R	G	B	A					
BC7_UNORM_SRGB	BDW				R	G	B	A					
BC6H_UF16	BDW				R	G	B	1.0*					
PLANAR_420_8	BDW				Cr	Y	Cb	1.0					
R8G8B8_UNORM_SRGB	BDW				R	G	B	1.0					
ETC1_RGB8	BDW				R	G	B	1.0					
ETC2_RGB8	BDW				R	G	B	1.0					
EAC_R11	BDW				R	0.0	0.0	1.0	OCL				
EAC_RG11	BDW				R	G	0.0	1.0	OCL				
EAC_SIGNED_R11	BDW				R	0.0	0.0	1.0	OCL				
EAC_SIGNED_RG11	BDW				R	G	0.0	1.0	OCL				

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
ETC2_SRGB8	BDW				R	G	B	1.0					
ETC2_RGB8_PTA	BDW				R	G	B	A					
ETC2_SRGB8_PTA	BDW				R	G	B	A					
ETC2_EAC_RGBA8	BDW				R	G	B	A					
ETC2_EAC_SRGB8_A8	BDW				R	G	B	A					

Programming Note

Context: SURFACE_STATE/Shader channel select

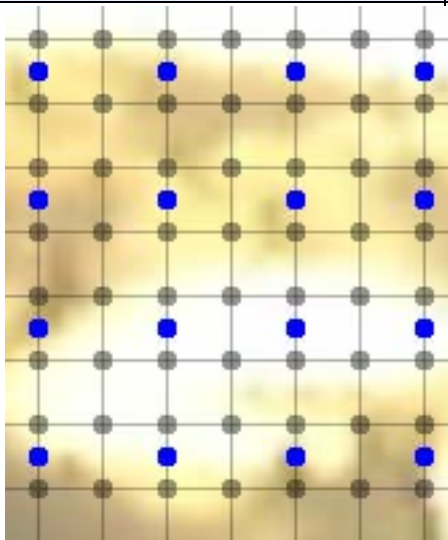
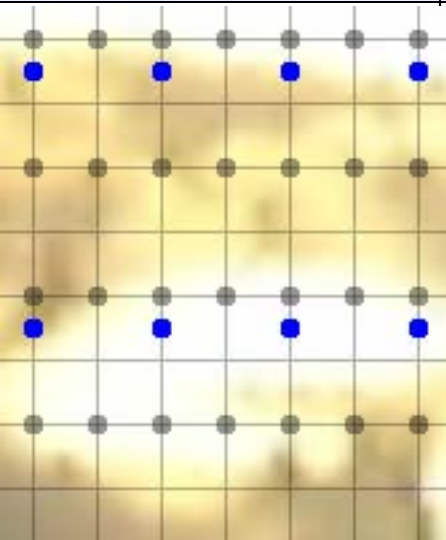
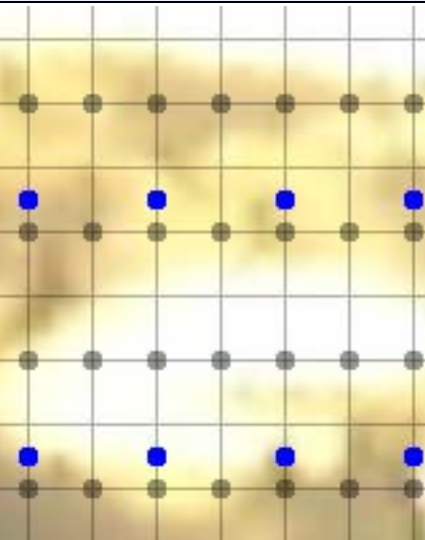
It is recommended, for performance reasons, to never use any format of the type L*A*, I* or A*. Instead use R* or RG* in combination with Shader Channel Select .

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

SAMPLER_STATE

SAMPLER_STATE has different formats, depending on the message type used. For BDW, 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_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

DWord	Bits	Description
	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
	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 R8_SINT R8G8B8_SINT
16 bpc, 32 bpt cases (4 types)	R16G16_UINT R16_UINT R16G16_SINT R16_SINT
32 bpc, 32 bpt case (2 types)	R32_UINT R32_SINT

HW supports only 1 index for a given Sampler Border Color state and Sampler State. So, SW will have to program the table in **SAMPLER_BORDER_COLOR_STATE** at DWord offsets 16 to 19, as per the integer surface format type (depends on the bits per channel and bits per texel of the surface format). If any color channel is missing from the surface format, the corresponding border color should be programmed as zero; if the alpha channel is missing, the corresponding Alpha border color should be programmed as 1. Some of the representative cases are listed below:

Case 2: R32G32B32A32_SINT (32 bpc, 128 bpt, 4 channel, SINT)

Each of the values in the above table would have to be programmed as sint32 value.

Case 3: R32G32B32_UINT (32 bpc, 128 bpt, 3 channel)

R/G/B values would be programmed like in Case1. Alpha channel value at DWORDN+3 would have to be programmed as Integer 1.

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 length of the response
- Message Header: Optional information which may be required for certain operations (e.g. Direct Write to Render Target)
- Message Payload: Specific data for each sampler operation including coordinates and other message parameters.

Programming Notes

Programming Note	
Context:	3D Sampler Messages
For GPGPU kernels which support mid-thread pre-emption must use a header on sample messages.	

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

Message Descriptor and Execution Mask

The following sub-sections explain the 16- and 8-bit execution masks and list the message descriptor register.

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

Message Descriptor - Sampling Engine

Message Header

The message header for the sampling engine is the same regardless of the message type. The message header is optional. If the header is not present, the behavior is as if the message was sent with all fields in the header set to zero and the write channel masks are all enabled and offsets are zero. However, if the header is not included in the message, the Sampler State Pointer will be obtained from the command stream input for the given thread. 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	31:0	Reserved
M0.4	31:0	Reserved
M0.3	31:5	Sampler State Pointer: Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for <i>ld</i> and <i>resinfo</i> message types. This pointer is relative to the Dynamic State Base Address . Format = DynamicStateOffset[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.
	4:0	Ignored
M0.2 spans so many rows, many for various projects, that the DWord value is repeated in each row.		
M0.2	31:22	Reserved
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.

DWord	Bits	Description						
		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).						
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. <table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2"><ul style="list-style-type: none">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.Alpha/Blue/Green/Red channels masked must set to 0 (no mask is supported).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.This field must be set to zero for all sample_min/sample_max instructions</td></tr></table>	Programming Note		Context:	3D Sampler Messages	<ul style="list-style-type: none">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.Alpha/Blue/Green/Red channels masked must set to 0 (no mask is supported).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.This field must be set to zero for all sample_min/sample_max instructions	
Programming Note								
Context:	3D Sampler Messages							
<ul style="list-style-type: none">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.Alpha/Blue/Green/Red channels masked must set to 0 (no mask is supported).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.This field must be set to zero for all sample_min/sample_max instructions								
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.						

DWord	Bits	Description												
		<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2"><ul style="list-style-type: none">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.This field must be set to zero for all sample_min/sample_max instructions</td></tr></table>	Programming Note		Context:	3D Sampler Messages	<ul style="list-style-type: none">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.This field must be set to zero for all sample_min/sample_max instructions							
Programming Note														
Context:	3D Sampler Messages													
<ul style="list-style-type: none">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.This field must be set to zero for all sample_min/sample_max instructions														
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	<p>U Offset: The u offset from the _aoffimmi modifier on the <i>sample</i> or <i>ld</i> 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.</p> <table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2"><ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> 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>	Programming Note		Context:	3D Sampler Messages	<ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> parameter is included in the gather4* messages.		Programming Note		Context:	Non-Normalized Floating-Point Coordinates	Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.	
Programming Note														
Context:	3D Sampler Messages													
<ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> parameter is included in the gather4* messages.														
Programming Note														
Context:	Non-Normalized Floating-Point Coordinates													
Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.														
M0.2	7:4	<p>V Offset: The v offset from the _aoffimmi modifier on the <i>sample</i> or <i>ld</i> 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.</p> <table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3DSampler Messages</td></tr><tr><td colspan="2"><ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> parameter is included in the gather4* messages.</td></tr></table>	Programming Note		Context:	3DSampler Messages	<ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> parameter is included in the gather4* messages.							
Programming Note														
Context:	3DSampler Messages													
<ul style="list-style-type: none">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.This field is ignored if the <i>offu</i> parameter is included in the gather4* messages.														

DWord	Bits	Description
		Programming Note
		Context: Non-Normalized Floating-Point Coordinates
		Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.
M0.2	3:0	R Offset: The r offset from the _aoffimmi modifier on the <i>sample</i> or <i>ld</i> 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: 3D Sampler Messages
		This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
		Programming Note
		Context: Non-Normalized Floating-Point Coordinates
		Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.
M0.1	31:0	Reserved
M0.0	31:0	Reserved

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.
_lz	LOD=0 Override: LOD is forced to 0. No LOD is calculated or provided in the message.
_d	Gradient: Rather than receiving absolute texel coordinates for all pixels of sub-span, a single pixel coordinate tuple is provided and a set of floating-point gradient values with respect to those pixel coordinates. This is then used to calculate other pixel coordinates and the LOD
_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.	

Sample Message Types

Sample Message Definition

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

Supported Variants

Message	Description
sample	Basic sample operation as described above.
sample_l	Basic sample with LOD in message, not computed
sample_b	Basic sample with Bias in message added to computed LOD.
sample_c	Basic sample with Reference value in message and comparison against Red channel of the sampled surface returned. Used primarily for shadow maps.
sample_lz	Basic sample with LOD forced to 0. Possibly higher performance than sample_l for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed or sent in message.
sample_l_c	sample_c with LOD override in message, not computed.
sample_b_c	sample_c with LOD Bias
sample_c_lz	Similar to sample_c with LOD forced to 0. Possibly higher performance than sample_c or sample_l_c for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed.
sample_d	<p>The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The <i>r</i> coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the <i>u</i>, <i>v</i>, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The <i>r</i> coordinate contains the faceid, and the <i>r</i> gradients are ignored by hardware.</p> <p>Previously known as sample_g.</p>
sample_d_c	Same as sample_d, but returns comparison against Red Channel of sampled surface like sample_c. Previously known as sample_g_c.
sample_killpix	Basic sample, but returns a Kill Pixel Mask based on Chroma Key sampler comparison results. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask.
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
<p>sample: The Surface Type of the associated surface must be SURFYPE_1D, SURFYPE_2D, SURFYPE_3D, or SURFYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. Sample is not supported in SIMD4x2 mode.</p>
<p>sample: Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p>

Restrictions and Programming Notes for sample_b

Programming Note
<p>sample_b: The Surface Type of the associated surface must be SURFYPE_1D, SURFYPE_2D, SURFYPE_3D, or SURFYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. sample_b is not supported in SIMD4x2 mode. 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.</p>

Restrictions and Programming Notes for sample_l and sample_lz

Programming Note
<p>sample_l and sample_lz: The Surface Type of the associated surface must be SURFYPE_1D, SURFYPE_2D, SURFYPE_3D, or SURFYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p>

Restrictions and Programming Notes for sample_c and sample_c_lz

Programming Note
<p>sample_c and sample_c_lz: The Surface Type of the associated surface must be SURFYPE_1D, SURFYPE_2D, or SURFYPE_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 <i>sample_c</i> 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. <i>sample_c</i> is not supported in SIMD4x2 mode. Using SURFYPE_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:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. Surface Type is SURFYPE_CUBE and any Cube Face Enable is disabled.

Programming Note

sample_c and sample_c_lz: The WGF sample_c_lz instruction is implemented by issuing the sample_c message with Force LOD to Zero enabled in the message header or by issuing the sample_l_c message with the LOD parameter set to zero.

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. However, unlike sample_c, sample_l_c is allowed as a SIMD4x2 message.

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.

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.

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. However, unlike sample_c, 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 disabled via the **Sampler Disable** bit in the SAMPLER_STATE, the Kill Pixel Mask returned will be undefined.

This message cannot be used with Fault & Stream mode.

Restrictions and Programming Notes for sample_min, sample_max

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D and Surface Array must be disabled.</p> <p>The Surface Format of the associated surface cannot be MONO8</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1</p>

Id Message Types

Supported Variants

Message	Description
Id	Basic Id operation as described above.
Id2dms	Basic Id operation but to a multi-sampled surface. Includes additional parameters for MCS values (see Id_mcs below) and Si (sample index) for each pixel in the message. Can be used for all multi-sampled surfaces up to X8 MSAA.
Id2dss	Same as Id2dms, 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.
Id_mcs	Basic Id 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 Id2dms/Id2dms_w/Id2dss message to sample from the multi-sampled surface.

Restrictions and Programming Notes for Id

Programming Note
<p>Id:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the Id message.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1</p>

Restrictions and Programming Notes for Id_mcs

Programming Note
Id_mcs: The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.
Id_mcs: If Id_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.
Id_mcs: The Id_mcs message uses the Auxiliary Surface Base Address and Auxiliary Surface Pitch fields in SURFACE_STATE to determine the base address and pitch of the surface. Surface Format is overridden to R8_UINT if Number of Multisamples is 2 or 4, R32_UINT if Number of Multisamples is 8, or R32G32_UINT if Number of Multisamples is 16. This message cannot be used on a non-multisampled surface. Otherwise, Id_mcs behaves like the Id message. If Id_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.

Restrictions and Programming Notes for Id2dms

Programming Note
Id2dms: The sampled surface type must not be MULTISAMPLECOUNT_1 or MULTISAMPLECOUNT_16. The Surface Type of the associated surface must be SURFTYPE_2D. The Surface Format of the associated surface cannot be MONO8.

gather4 Message Types

gather4 Message Type Description

Message Type	Description				
gather4_*	<p>The surface is sampled using MAP_LINEAR filtering, regardless of the filtering mode specified in the sampler state and with LOD forced to zero of the visible resource.</p> <p>However, the samples are not filtered. Instead, the selected color channel the four contributing texels are returned directly in the sample's corresponding four channels as follows:</p> <table border="1"> <tr> <td>upper left sample = alpha channel</td><td>upper right sample = blue channel</td></tr> <tr> <td>lower left sample = red channel</td><td>lower right sample = green channel</td></tr> </table> <p>Returned color channel is selected through a message field: Gather4 Source Channel Select</p> <p>Two or three floating-point coordinates may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters default to 0.</p>	upper left sample = alpha channel	upper right sample = blue channel	lower left sample = red channel	lower right sample = green channel
upper left sample = alpha channel	upper right sample = blue channel				
lower left sample = red channel	lower right sample = green channel				

Supported Variants:

Message	Description
gather4	Basic gather4 behavior as described above.
gather4_c	Similar to basic gather4 but performs a compare between a Reference parameters and the gathered pixels and returns a white or black (1 or 0). In addition, like sample_c , it only returns data on the Red channel, so the Gather4 Source Channel Select must be set to Red.
gather4_po	Similar to gather4, but includes offu and offv parameters, which contain texel-space offsets that override the U/V Offset fields in the message header. Unlike the message header fields however, these offsets have a wider range [-32,+31], and can differ per pixel or sample. The format of the data is 32-bit 2's complement signed integer, but hardware only interprets the least significant 6 bits of each value, treating it as a 6-bit 2's complement signed integer
gather4_po_c	This is a combination of gather4_c and gather4_po, and will return data only on the Red Channel (Gather4 Source Channel Select must be set to Red), but uses offu and offv to shift the position on the surface.

Restrictions and Programming Notes for gather4:

Programming Note
<p>gather4: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode when either of the following is true will yield undefined results:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.
<p>gather4: If Surface Format is R10G10B10_SNORM_A2_UNORM and Gather4 Source Channel Select is alpha channel, the returned value may be incorrect.</p>

Restrictions and Programming Notes for gather4_c:

Programming Note
<p>gather4_c: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE.</p> <p>The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, or R16_UNORM.</p> <p>The channel selected by the Gather4 Source Channel Select field in the message header must be set to Red.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.

Restrictions and Programming Notes for gather4_po:

Programming Note
<p>gather4_po: The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. <p>Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.</p>
<p>gather4_po: The surface position using offu/offv are calculated in normalized space and hence subject to small truncation error.</p>
<p>gather4_po: If Surface Format is R10G10B10_SNORM_A2_UNORM and Gather4 Source Channel Select is alpha channel, the returned value may be incorrect.</p>
<p>gather4_po: offu/offv are calculated in normalized space and hence subject to small truncation error.</p>

Restrictions and Programming Notes for gather4_po_c:

Programming Note
<p>gather4_po_c: The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, or R16_UNORM</p> <p>The channel selected by the Gather4 Source Channel Select field in the message header must be set to Red.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.
<p>gather4_po_c: The surface position using offu/offv are calculated in normalized space and hence subject to small truncation error.</p>

sampleinfo Message Type

sampleinfo Message Definition

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

Supported Variants:

None

LOD Message Type

LOD Message Definition

Message Type	Description
LOD	<p>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.</p>

Supported Variants:

None

Restrictions and Programming Notes for LOD:

Programming Note
<p>LOD:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.</p> <p>The Surface Format of the associated surface cannot be MONO8.</p> <p>LOD is not supported in SIMD4x2 mode.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p>

resinfo Message Type

resinfo Message Definition

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

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	(Height+1)»LOD	Surface Array ? Depth+1 : 0	MIP Count
SURFTYPE_3D	(Width+1)»LOD	(Height+1)»LOD	(Depth+1)»LOD	MIP Count
SURFTYPE_CUBE	(Width+1)»LOD	(Height+1)»LOD	Surface Array ? Depth+1 : 0	MIP Count
SURFTYPE_BUFFER SURFTYPE_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.

cache_flush Message Type

cache_flush Message Definition

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

Supported Variants:

None

Restrictions and Programming Notes for cache_flush:

Programming Note
<p>cache_flush:</p> <p>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.</p> <p>This message must be sent as a SIMD32/64 type message with header only (no additional message contents)</p>
<p>cache_flush:</p> <p>The cache_flush message cannot be used.</p>
<p>cache_flush:</p> <p>The STunit Power Optimization Disable bit in the HALF_SLICE_CHICKEN2 (E180h) must be programmed to 1h to utilize this message.</p>

Media Message Types

The 3D sampler supports specific message types which are intended for use by the Media sampler. These messages include:

Message
sample_unorm
sample_8X8

Media message types are described in more detail, with payload information defined in the Media Sampler Section.

sample_unorm Message Types

sample_unorm Message Definition

Message Type	Description
sample_unorm_*	<p>The surface is sampled using the indicated sampler state. 32 contiguous pixels in a 8-wide by 4-high block are sampled. The U and V addresses for the upper left pixel are delivered in this message along with Delta U and Delta V floating-point parameters and . Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:</p> $U(x,y) = U(0,0) + \text{DeltaU} * x$ $V(x,y) = V(0,0) + \text{DeltaV} * y$ <p>Typically this message and its variants are used for media applications.</p>

Supported Variants:

Message	Description
sample_unorm	Basic sample_unorm behavior as described above.
sample_unorm_RG	Similar to sample_unorm, but only the Red and Green Channels are returned because there is an implied write mask.
sample_unorm_killpix	This message is identical to the sample_unorm message except it returns a kill pixel mask in addition to the selected channels in the writeback message. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. All restrictions of the sample_unorm message apply to this message also.
sample_unorm_RG_killpix	This message is identical to the sample_unorm_RG message except it returns a kill pixel mask in addition to the selected channels in the writeback message. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. All restrictions of the sample_unorm message apply to this message also.

Restrictions and Programming Notes for sample_unorm, sample_unorm_RG, sample_unorm_killpix, sample_unorm_RG_killpix:

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface must be UNORM with ≤ 8 bits per channel.</p> <p>The MIP Count, Depth, Surface Min LOD, Resource Min LOD, and Min Array Element of the associated surface must be 0.</p> <p>The Min and Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR.</p> <p>The Mip Mode Filter must be MIPFILTER_NONE.</p> <p>The TCX and TCY Address Control Mode cannot be any of:</p> <ul style="list-style-type: none"> • TEXCOORDMODE_CLAMP_BORDER • TEXCOORDMODE_HALF_BORDER • TEXCOORDMODE_MIRROR_ONCE • TEXCOORDMODE_WRAP <p>Map Width must be ≥ 4.</p> <p>Map Height must be ≥ 4</p> <ul style="list-style-type: none"> • Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
Both $\Delta U \cdot \text{Width}$ and $\Delta V \cdot \text{Height}$ of the associated surface must be less than or equal to 3.0.
The TCX and TCY Control Mode cannot be TEXCOORDMODE_MIRROR.

sample_8x8 Message Type

Supported Variants:

None.

Restrictions and Programming Notes for sample_8x8:

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface must be UNORM with ≤ 10 bits per channel.</p> <p>$\Delta V * Height$ of the associated surface must be less than 16.0.</p> <ul style="list-style-type: none"> $Map Width$ must be ≥ 4.
<p>Supports functionality such as convolve, MinMax, MinMaxFilter, Dilate, Erode, BoolCentroid, and Centroid have been added on top of the existing sample_8x8. The convolve, MinMax, and MinMaxFilter are 16-wide X 4-high; the rest have variable size depending on the message.. See Media Sampler section for a description of how these modes work.</p>

Message Format

The following section describes how messages are formatted, including how message length and response length are calculated. The sampler supports a variety of different message types which perform different sampling and filtering operations (see Message Types section for a description of these sampler and filtering operations). For each message type, different SIMD modes are supported which determines how many pixels the sampler performs the specified operation on in parallel. The precision of the input parameters and resulting samples can also be pr

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

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 ("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. 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	$H + N$
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^{**}
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.

* 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.

Programming Note	
Context:	3D Sampler Messages - Message Format
Parameter 0 is required except for the sampleinfo message, which has no parameter 0.	

Message Formats

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

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

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

SIMD	Message Type	Mnemonic	Parameters										
			0	1	2	3	4	5	6	7	8	9	10
SIMD8, SIMD16	00000	sample	u	v	r	ai							
	00001	sample_b	bias	u	v	r	ai						
	00010	sample_l	lod	u	v	r	ai						
	00011	sample_c	ref	u	v	r	ai						
	00100	sample_d	u	dudx	dudy	v	dvdv	dvdv	r	drdx	drdy	ai	
	00101	sample_b_c	ref	bias	u	v	r	ai					
	00110	sample_l_c	ref	lod	u	v	r	ai					
	00111	ld	u	lod	v	r							
	01000	gather4	u	v	r	ai							
	01001	LOD	u	v	r	ai							
	01010	resinfo	lod										
	01011	sampleinfo											
	01100	sample + killpix	u	v	r								
	10000	gather4_c	ref	u	v	r	ai						
	10001	gather4_po	u	v	offu	offv	r						
	10010	gather4_po_c	ref	u	v	offu	offv	r					
	10100	sample_d_c	ref	u	dudx	dudy	v	dvdv	dvdv	r	drdx	drdy	ai
	10110	sample_min	u	v									
	10111	sample_max	u	v									
	11101	ld_mcs	u	v	r								
	11110	ld2dms	si	mcs	u	v	r						
	11111	ld2dss	ssi	u	v	r							

SIMD	Message Type	Mnemonic	Parameters										
			0	1	2	3	4	5	6	7	8	9	10
SIMDD4X2	00010	sample_l	u	v	r	ai	lod						
	00100	sample_d	u	v	r	ai	dudx	dudy	dvdxd	dvdxdy	drdx	drdy	
	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	dvdxd	dvdxdy	drdx	drdy	ref
	11101	ld_mcs	u	v	r								
	11110	ld2dms	u	v	r		si	mcs					
SIMD32, SIMD64	00000	sample_unorm											
	00010	sample_unorm+killpix											
	01000	deinterlace											
	01100	sample_unorm											
	01010	sample_unorm +killpix											
	01011	sample_8x8											
	11111	cache_flush											

Programming Note

Context: 3D Sampler Messages - Message Format

For output control modes other than "16 bit Full", the response lengths are reduced accordingly. Refer to the writeback message format definitions to determine the correct response length for each case.

Programming Note

Context: 16-bit sample returns

When a sample results in a denorm value which is to be returned in 16-bit floating-point, the lsb of the value may be incorrect due to rounding errors.

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.

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 Id and resinfo message types.
M1.6	31:0	Subspan 1, Pixel 2 (lower left) Parameter 0
M1.5	31:0	Subspan 1, Pixel 1 (upper right) Parameter 0
M1.4	31:0	Subspan 1, Pixel 0 (upper left) Parameter 0
M1.3	31:0	Subspan 0, Pixel 3 (lower right) Parameter 0
M1.2	31:0	Subspan 0, Pixel 2 (lower left) Parameter 0
M1.1	31:0	Subspan 0, Pixel 1 (upper right) Parameter 0
M1.0	31:0	Subspan 0, Pixel 0 (upper left) Parameter 0
M2.7	31:0	Subspan 3, Pixel 3 (lower right) Parameter 0
M2.6	31:0	Subspan 3, Pixel 2 (lower left) Parameter 0
M2.5	31:0	Subspan 3, Pixel 1 (upper right) Parameter 0
M2.4	31:0	Subspan 3, Pixel 0 (upper left) Parameter 0
M2.3	31:0	Subspan 2, Pixel 3 (lower right) Parameter 0
M2.2	31:0	Subspan 2, Pixel 2 (lower left) Parameter 0
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 Id and resinfo message types.
M1.6	31:0	Subspan 1, Pixel 2 (lower left) Parameter 0
M1.5	31:0	Subspan 1, Pixel 1 (upper right) Parameter 0
M1.4	31:0	Subspan 1, Pixel 0 (upper left) Parameter 0
M1.3	31:0	Subspan 0, Pixel 3 (lower right) Parameter 0
M1.2	31:0	Subspan 0, Pixel 2 (lower left) Parameter 0
M1.1	31:0	Subspan 0, Pixel 1 (upper right) Parameter 0
M1.0	31:0	Subspan 0, Pixel 0 (upper left) Parameter 0
M2 – Mn		Repeat packet 1 to cover all required parameters.

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 Id 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 (SIMD16, SIMD8, SIMD4x2, or SIMD32/64).

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.

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. Format depends on the Data Return Format programmed for the surface being sampled.
W0.6	31:0	Subspan 1, Pixel 2 (lower left) Red
W0.5	31:0	Subspan 1, Pixel 1 (upper right) Red
W0.4	31:0	Subspan 1, Pixel 0 (upper left) Red
W0.3	31:0	Subspan 0, Pixel 3 (lower right) Red
W0.2	31:0	Subspan 0, Pixel 2 (lower left) Red
W0.1	31:0	Subspan 0, Pixel 1 (upper right) Red
W0.0	31:0	Subspan 0, Pixel 0 (upper left) Red
W1.7	31:0	Subspan 3, Pixel 3 (lower right) Red
W1.6	31:0	Subspan 3, Pixel 2 (lower left) Red
W1.5	31:0	Subspan 3, Pixel 1 (upper right) Red
W1.4	31:0	Subspan 3, Pixel 0 (upper left) Red
W1.3	31:0	Subspan 2, Pixel 3 (lower right) Red
W1.2	31:0	Subspan 2, Pixel 2 (lower left) Red
W1.1	31:0	Subspan 2, Pixel 1 (upper right) Red
W1.0	31:0	Subspan 2, Pixel 0 (upper left) Red
W2		Subspans 1 and 0 of Green: See W0 definition for pixel locations
W3		Subspans 3 and 2 of Green: See W1 definition for pixel locations
W4		Subspans 1 and 0 of Blue: See W0 definition for pixel locations
W5		Subspans 3 and 2 of Blue: See W1 definition for pixel locations
W6		Subspans 1 and 0 of Alpha: See W0 definition for pixel locations

DWord	Bit	Description
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
	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

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. Format depends on the Surface Format programmed for the surface being sampled.
	15:0	Subspan 3, Pixel 2 (lower left) Red
W0.6	31:16	Subspan 3, Pixel 1 (upper right) Red
	15:0	Subspan 3, Pixel 0 (upper left) Red
W0.5	31:16	Subspan 2, Pixel 3 (lower right) Red
	15:0	Subspan 2, Pixel 2 (lower left) Red
W0.4	31:16	Subspan 2, Pixel 1 (upper right) Red
	15:0	Subspan 2, Pixel 0 (upper left) Red
W0.3	31:16	Subspan 1, Pixel 3 (lower right) Red
	15:0	Subspan 1, Pixel 2 (lower left) Red
W0.2	31:16	Subspan 1, Pixel 1 (upper right) Red
	15:0	Subspan 1, Pixel 0 (upper left) Red
W0.1	31:16	Subspan 0, Pixel 3 (lower right) Red
	15:0	Subspan 0, Pixel 2 (lower left) Red
W0.0	31:16	Subspan 0, Pixel 1 (upper right) Red
	15:0	Subspan 0, Pixel 0 (upper left) Red
W1		Green: See W0 definition for pixel locations
W2		Blue: See W0 definition for pixel locations
W3		Alpha: See W0 definition for pixel locations
W4.7:1		Reserved (not written): W8 is only delivered when Pixel Fault Mask Enable is enabled.
W4.0	31:16	Reserved: always written as 0xffff
	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/SIMD8D

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.

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. Format depends on the Data Return Format programmed for the surface being sampled.
W0.6	31:0	Subspan 1, Pixel 2 (lower left) Red
W0.5	31:0	Subspan 1, Pixel 1 (upper right) Red
W0.4	31:0	Subspan 1, Pixel 0 (upper left) Red
W0.3	31:0	Subspan 0, Pixel 3 (lower right) Red
W0.2	31:0	Subspan 0, Pixel 2 (lower left) Red
W0.1	31:0	Subspan 0, Pixel 1 (upper right) Red
W0.0	31:0	Subspan 0, Pixel 0 (upper left) Red
W1		Subspans 1 and 0 of Green: See W0 definition for pixel locations
W2		Subspans 1 and 0 of Blue: See W0 definition for pixel locations
W3		Subspans 1 and 0 of Alpha: See W0 definition for pixel locations
W4.7:1		Reserved (not written) : This W4 is only delivered for the sample+killpix message type
W4.0	31:16	Dispatch Pixel Mask: This field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread.
	15:0	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. 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.
W4.7:1		Reserved (not written): This W4 is only delivered when Pixel Fault Mask Enable is enabled.
W4.0	31:8	Reserved: always written as 0xffffffff
	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.

Return Format = 16-bit

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. Format depends on the Surface Format programmed for the surface being sampled.
	15:0	Subspan 1, Pixel 2 (lower left) Red
W0.2	31:16	Subspan 1, Pixel 1 (upper right) Red
	15:0	Subspan 1, Pixel 0 (upper left) Red
W0.1	31:16	Subspan 0, Pixel 3 (lower right) Red
	15:0	Subspan 0, Pixel 2 (lower left) Red
W0.0	31:16	Subspan 0, Pixel 1 (upper right) Red
	15:0	Subspan 0, Pixel 0 (upper left) Red
W1		Subspans 1 and 0 of Green: See W0 definition for pixel locations
W2		Subspans 1 and 0 of Blue: See W0 definition for pixel locations
W3		Subspans 1 and 0 of Alpha: See W0 definition for pixel locations
W4.7:1		Reserved (not written): This W4 is only delivered when Pixel Fault Mask Enable is enabled.
W4.0	31:8	Reserved: always written as 0xffffffff
	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.

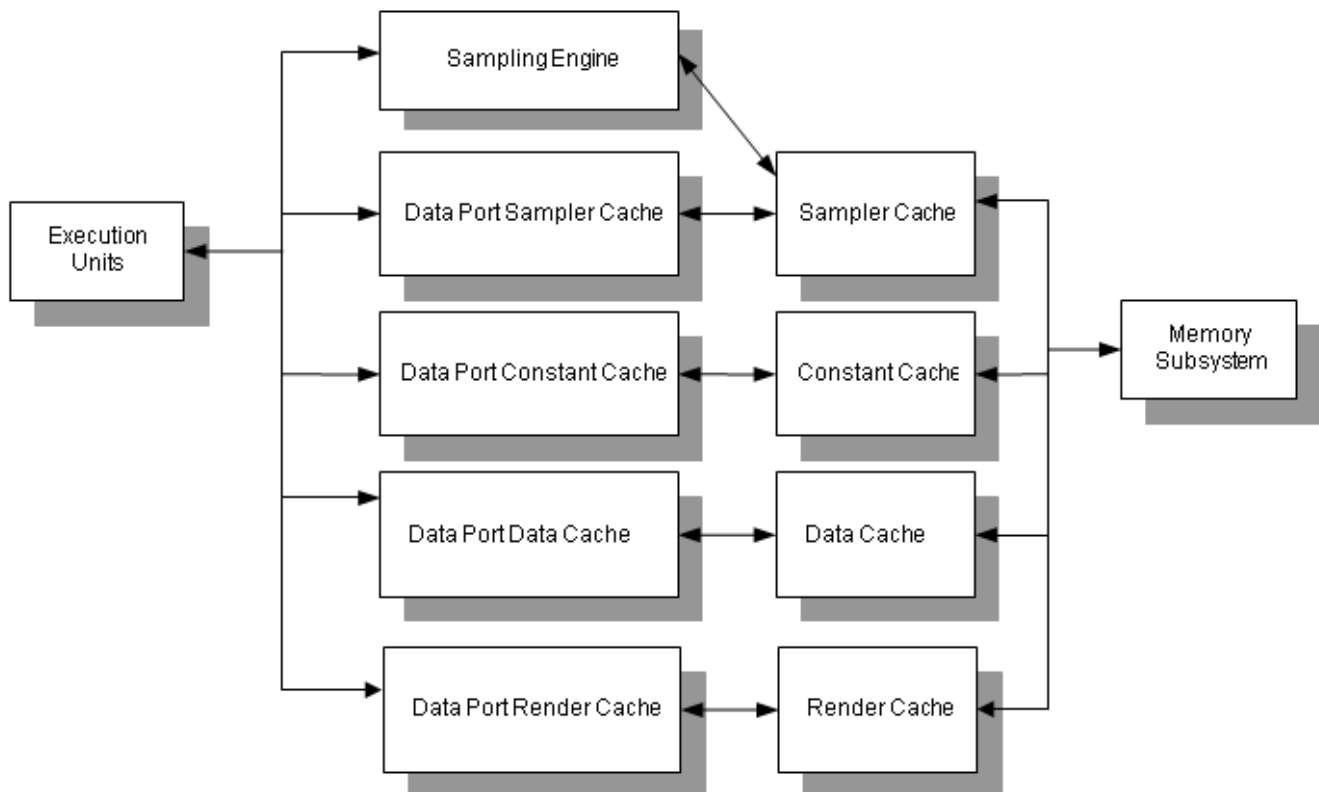
DWord	Bit	Description
W0.7	31:0	Sample 1 Alpha: Specifies the value of the pixel's alpha channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the Data Return Format programmed for the surface being sampled.
W0.6	31:0	Sample 1 Blue
W0.5	31:0	Sample 1 Green
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) : W4 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.

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 render target writes, constant buffer reads, scratch space reads/writes, and media surface accesses.

BDW adds the Data Port Data Cache and the Data Cache.

The diagram below shows the four parts of the Data Port (Sampler Cache, Constant Cache, Data Cache, and Render Cache) and how they connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.



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 four data ports are considered to be separate shared functions, each with its own shared function identifier.

Data Cache

The data cache is a read/write cache that is coherent across the physical instances of this cache. It is intended to be used for the following surfaces:

- constant buffers
- destination surfaces for media applications
- intermediate working surfaces for media applications
- scratch space buffers
- general read/write access of surfaces
- atomic operations
- shared memory for GPGPU thread groups

The data cache can be accessed via the *Data Cache Data Port* shared function, and via the load and store EU messages. Ordering from a single thread is maintained when accessing the data cache using only one of these mechanisms, but is not maintained when using both of these mechanisms from the same thread. In these instances, software must ensure ordering by using write commits and/or waiting for read data to be returned.

Data Cache Coherency

Data cache memory is backed by system memory. The read/write data cache is always coherent between GPU thread accesses, but might not be coherent with CPU memory accesses.

Coherency Type
<p>Two types of coherency are supported for data cache memory: GPU Coherent and IA Coherent. GPU coherent accesses are kept coherent with GPU threads but not kept coherent with CPU (LLC) accesses. IA Coherent accesses are kept coherent with both GPU threads and with CPU accesses.</p> <p>Two GPU accesses made to the same address but with different coherency types have different cache entries and are not kept coherent with each other. If a buffer will be accessed with different coherency types, software is responsible for flushing the GPU cache to system memory, using either a Memory Fence message or a pipeline flush command.</p> <p>The surface state field Coherency Type specifies the data cache coherency used with that surface. Stateless accesses specify their coherency with BTI=255 or BTI=253.</p> <p>To support legacy behavior, all IA Coherent accesses are overridden to be GPU coherent if the HDC_CHICKEN0 register Force Non-Coherent control is set.</p> <p>During preemption when the SIP routine is used save and restore registers, if the Force Context Save/Restore Non-Coherent control is set in the HDC_CHICKEN0 register, then the registers will forced to be saved and restored using GPU coherent accesses.</p>

Sampler Cache

The sampler cache is a read-only cache that supports both linear and tiled memory. In addition to being used by the sampling engine (via the sampling engine messages), the sampler cache is intended to be used for source surfaces in media applications via the data port. The same application may use the sampler cache via the sampling engine and data port without flushing the pipeline between accesses.

Surfaces

The data elements accessed by the data port are called “surfaces”. There are two models used by the data port to access these surfaces: surface state model and stateless model.

Surface State Model

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

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

Stateless Model

The stateless model is used when a **Binding Table Index** (specified in the message descriptor) of 255 is specified.

This model is primarily intended to be used for scratch space buffers.

In this 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 = R32G32B32A32_FLOAT
- Vertical Line Stride = 0
- Surface Base Address = General State Base Address + Immediate Base Address
- Surface Pitch = 16 bytes
- Utilize Fence = false
- Tiled = false

Buffer Size Checking

Buffer Size Checking
Buffer Size = checked only against General State Buffer Size When General State Buffer Size is zero, any stateless access is out-of-bounds.

Shared Local Memory (SLM)

The 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 a portion of its space for the SLM. One SLM is present in each half slice, and its contents are shared between all of the active threads in that half slice. Its contents are uninitialized after creation, and its contents disappear when deallocated.

The SLM is accessed when a **Binding Table Index** (specified in the message descriptor) of 254 is specified. 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. This includes the Byte Scattered Read/Write, Untyped Surface Read/Write, and Untyped Atomic Operation messages. In addition, only the data cache data port is supported; the other data ports treat Binding Table Index 254 as a normal surface state access.

Programming Note	
Context:	Shared Local Memory (SLM)
Accesses to SLM don't have any bounds checking. Addresses beyond the size (64KB) of the SLM wrap around.	

Write Commit

For write messages, an optional write commit writeback message can be requested via the Send Write Commit Message bit in the message descriptor. This bit causes a return message to the thread indicating when the write has been committed to the in-order cache pipeline and it is safe to issue another access to the same data with the assurance that it will happen after the first write. A read issued after the write commit ensures that the read will get the newly written data, and another write issued after the write commit will be the last to modify the data. "Committed" does not guarantee that the data has been actually written to the memory subsystem, but only that the write has been scheduled and cannot be passed by another read or write issued subsequently.

If **Send Write Commit Message** is used on a Flush Render Cache message, the write commit is sent only when the render cache has completed its flush to memory. A read issued to another cache after the write commit is received will be guaranteed to retrieve the "new" data that was written before the Flush Render Cache message was issued.

The write commit does not modify the destination register, but merely clears the dependency associated with the destination register. Thus, a simple "mov" instruction using the register as a source is sufficient to wait for the write commit to occur. The following code sequence indicates this:

```
send r12 m1 DPWRITE      ; Issue write to render cache.
mov m1 r3                ; Assemble read message.
mov r12 r12              ; Block on write commit.
send r13 m1 DPREAD       ; Read same location as write.
```

ReadWrite Ordering

Reads and writes issued from the same thread *are* guaranteed to be processed in the same order as issued. Software mechanisms must still ensure any needed ordering of accesses issued from different threads.

Accessing Buffers

There are four data port messages used to access buffers. Three of these are used for both constant buffers and scratch space buffers, the fourth is used by the geometry shader kernel to write to streamed vertex buffers. All of these messages support only buffers, and can use the surface state model as well as the stateless model.

The following table indicates the intended applications of each of the buffer messages.

Message	Applications
OWord Block Read/Write	<ul style="list-style-type: none"> 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
OWord Dual Block Read/Write	<ul style="list-style-type: none"> SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access) SIMD4x2 scratch space reads/writes where the indices are different.
DWord Scattered Read/Write	<ul style="list-style-type: none"> 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
Streamed Vertex Buffer Write	<ul style="list-style-type: none"> geometry shader streaming vertex data out

These messages generally ignore the surface format field of the state and perform no format conversion. The exception is the Streamed Vertex Buffer Write, which uses the surface format field to determine only how many channels are to be written. The data contained in each channel is still not converted in any way.

Accessing Media Surfaces

The Media Block Read/Write message is intended to be used to access 2D media surfaces. The message specifies an X/Y coordinate into the 2D surface as input. Since this message only supports 2D surfaces, the stateless model cannot be used with this message.

Boundary Behavior

The table below summarizes the behavior of the **Media Boundary Pixel Mode** field (SURFACE_STATE) in combination with the **Vertical Line Stride** and **Vertical Line Stride Offset** fields (both of which are subject to being overridden by the Data Port message descriptor fields). The Behavior column illustrates behavior for a surface with four rows numbered 0 to 3. The bold indicators are off-surface behavior and the non-bold indicators are on-surface behavior. Input row addresses range from -3 to +7 going left to right.

Media Boundary Pixel Mode	Vertical Line Stride	Vertical Line Stride Offset	Usage Model	Behavior
0	0	X	normal frame	000001233333
0	1	0	normal field even	000002222222
0	1	1	normal field odd	111113333333
2	0	X	frame / progressive	000001233333
2	1	0	field even / progressive	000002333333
2	1	1	field odd / progressive	000013333333
3	0	X	frame / interlaced	010101232323
3	1	0	field even / interlaced	000002222222
3	1	1	field odd / interlaced	111113333333

State

This topic discusses applicable State options for the Data Port.

BINDING_TABLE_STATE

The data port uses the binding table to retrieve surface state. Refer to [State](#) in the Sampling Engine section for the definition of this state.

SURFACE_STATE

The data port uses the surface state for constant buffers, render targets, and media surfaces.

Messages

This section of the PRM discusses messages applicable to the Data Port.

Global Definitions

For data port messages, part of the message descriptor is used to determine the message type. This field is documented here. The remainder of the message descriptor is defined differently depending on the message type, and is documented in the section for the corresponding message.

The Data Port is actually separate targets, **Data Port Sampler Cache**, **Data Port Constant Cache**, and **Data Port Render Cache**, each with its own target unit ID. Each target has its own set of message type encodings as shown below.

Programming Note	
Context:	Messages
<p>Data port messages may not have the End of Thread bit set in the message descriptor other than the following exceptions:</p> <ul style="list-style-type: none"> The Render Target Write message may have End of Thread set for pixel shader threads dispatched by the windower in non-contiguous dispatch mode. The Render Target UNORM Write message may have End of Thread set for pixel shader threads dispatched by the windower in contiguous dispatch mode. The Media Block Write message may have End of Thread set for pixel shader threads dispatched by the windower in contiguous dispatch mode. 	

Data Port Messages

The table below summarizes the data port messages and address modes they support.

Data Cache Data Port Message Summary

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
OWord Block Read	yes	no	yes	global	1
OWord Block Write	yes	no	yes	global	1
Unaligned OWord Block Read	yes	no	yes	global	1
OWord Dual Block Read	no for stated yes for stateless	no	yes	global + offset	2
OWord Dual Block Write	no for stated yes for stateless	no	yes	global + offset	2
A64 QWord Scattered Read	no (forbidden)	no	yes (only)	offset	8
A64 QWord Scattered Write	no (forbidden)	no	yes (only)	offset	8
A64 DWord Scattered Read	no (forbidden)	no	yes (only)	offset	8
A64 DWord Scattered Write	no (forbidden)	no	yes (only)	offset	8
A64 Byte Scattered Read	no (forbidden)	no	yes (only)	offset	8
A64 Byte Scattered Write	no (forbidden)	no	yes (only)	offset	8
A64 Untyped Surface Read	no (forbidden)	no	yes (only)	offset	2, 8
A64 Untyped Surface Write	no (forbidden)	no	yes (only)	offset	2, 8
A64 Untyped Atomic Operation	no (forbidden)	no	yes (only)	offset	8
A64 Untyped Atomic Operation SIMD4x2	no (forbidden)	no	yes (only)	offset	2
A64 Block Read/Write	yes	no	yes (only)	offset	1
DWord Scattered Read	no for stated yes for stateless	no	yes	global + offset	8, 16
DWord Scattered Write	no for stated yes for stateless	no	yes	global + offset	8, 16
Byte Scattered Read	no for stated yes for stateless	yes	yes	global + offset	8, 16
Byte Scattered Write	no for stated yes for stateless	yes	yes	global + offset	8, 16

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
Untyped Surface Read	no for stated yes for stateless	yes	yes (1D only)	1D or 2D	2, 8, 16
Untyped Surface Write	no for stated yes for stateless	yes	yes (1D only)	1D or 2D	2, 8, 16
Untyped Atomic Operation	no for stated yes for stateless	yes	yes (1D only)	1D or 2D	8, 16
Untyped Atomic Operation SIMD4x2	no for stated yes for stateless	yes	yes (1D only)	1D or 2D	2
Atomic Counter Operation	no ² Required for inc, dec, predec	no	no	implied	8
Atomic Counter Operation SIMD4x2	no ² Required for inc, dec, predec	no	no	implied	2
Scratch Block Read	yes	no	yes (only)	Imm_Buf + offset	
Scratch Block Write	yes	no	yes (only)	Imm_Buf + offset	
Memory Fence	yes	N/A	N/A	N/A	N/A
Typed Surface Read	yes	no	no	1D, 2D, 3D, 4D	2, 8
Typed Surface Write	yes	no	no	1D, 2D, 3D, 4D	2, 8
Typed Atomic Operation	yes	no	no	1D, 2D, 3D, 4D	8
Typed Atomic Operation SIMD4x2	yes	no	no	1D, 2D, 3D, 4D	2, 8
Media Block Read	yes	no	no	2D	1
Media Block Write	yes	no	no	2D	1
Transpose Read	yes	no	no	2D	1

Table Notes

"global" is the **Global Offset** in the message header (if header is not present, Global Offset is zero).

"imm_buf" is the Immediate Buffer Base Address provided in message header register M0.5.

"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.

"1D" and "2D" are the address payload.

Render Cache Data Port Message Summary

Message Type	Header Required	Address Modes	Vector Width
Render Target Write	no	2D + RTAI	8, 16

Message Descriptor

This section describes the Data Port message descriptors for various projects.

Message Descriptor

SAMPLER CACHE DATA PORT		RENDER CACHE DATA PORT	
Bit	Description	Bit	Description
19	Header Present. If set, indicates that the message includes the header. Note: For the Sampler Cache Data Port, the header must be present for the following message types: Unaligned OWord Block Read Media block read Format = Enable		
18	Ignored	18	Ignored
17:14	Message Type 0000: Read Surface Info 0001: Unaligned OWord Block Read 0100: Media Block Read All other encodings are reserved.	17:14	Message Type 1100: Render Target Write 1101: Render Target Read All other encodings are reserved.
13:8	Message Specific Control. Refer to the specific message section for the definition of these bits.		
7:0	Binding Table Index. Specifies the index into the binding table for the specified surface. Format = U8 Range = [0,255]		

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT0		DATA CACHE DATA PORT1	
Bit	Description	Bit	Description	Bit	Description
19	Header Present. If set, indicates that the message includes the header. Programming Note: For the Data Cache Data Port*, the header must be present for the following message types: <ul style="list-style-type: none"> • OWord Block Read/Write • Unaligned OWord Block Read • Memory Fence • Scratch read/write • Typed read/write/atomics • Media block read/write 				

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT0		DATA CACHE DATA PORT1	
Bit	Description	Bit	Description	Bit	Description
	<ul style="list-style-type: none"> • Transpose Read • A64 Block Read/Write <p>For the Constant Cache Data Port, the header must be present for the following message types:</p> <ul style="list-style-type: none"> • OWord Block Read/Write • Unaligned OWord Block Read • Format = Enable 				
18	Ignored	18	Category 1: Scratch Block Read/Write messages 0: Legacy DAP-DC messages	18:14	

17:14	Message Type 0000: OWord Block Read 0001: Unaligned OWord Block Read 0010: OWord Dual Block Read 0011: DWord Scattered Read All other encodings are reserved.	17:14	Category=0 (legacy dataport) Message Type 0000: OWord Block Read 0001: Unaligned OWord Block Read 0010: OWord Dual Block Read 0011: DWord Scattered Read 0100: Byte Scattered Read 0111: Memory Fence 1000: OWord Block Write 1010: OWord Dual Block Write 1011: DWord Scattered Write 1100: Byte Scattered Write All other encodings are reserved. Category=1 (scratch) [17]: 0=Read; 1=write [16]: Type; 0=Oword, 1=Dword [15]: Invalidate after read; [14]: <Reserved, mbz> [13:12]: Block Size 11: 8 registers 10: 4 registers 01: 2 registers 00: 1 register [11:0]: Addr offset (Hword based)	Message Type 00000: Transpose Read 00001: Untyped Surface Read 00010: Untyped Atomic Operation 00011: Untyped Atomic Operation SIMD4x2 00100: Media Block Read 00101: Typed Surface Read 00110: Typed Atomic Operation 00111: Typed Atomic Operation SIMD4x2 01000: Untyped Atomic Float Add 01001: Untyped Surface Write 01010: Media Block Write 01011: Atomic Counter Operation 01100: Atomic Counter Operation SIMD4x2 01101: Typed Surface Write 10000: A64 Scattered Read 10001: A64 Untyped Surface Read 10010: A64 Untyped Atomic Operation 10011: A64 Untyped Atomic Operation SIMD4x2 10100: A64 Block Read 10101: A64 Block Write 11000: A64 Untyped Atomic Float Add 11001: A64 Untyped
-------	---	-------	---	--

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT0		DATA CACHE DATA PORT1	
Bit	Description	Bit	Description	Bit	Description
					Surface Write 11010: A64 Scattered Write All other encodings are reserved.
13:8	Message Specific Control. Refer to the specific message section for the definition of these bits.				
7:0	Binding Table Index. Specifies the index into the binding table for the specified surface. For the data cache data port, two binding table indexes are used to select special surfaces: 254: A binding table index of 254 indicates that the shared local memory (SLM) is to be used. The SLM is only supported with the Byte Scattered Read/Write, Untyped Surface Read/Write, and Untyped Atomic Operation messages. Refer to the <i>Shared Local Memory</i> section earlier in this chapter for further details on its behavior. 255: A binding table index of 255 indicates that a stateless model is to be used. Refer to the <i>Stateless Model</i> section for details. 253: An alias for Stateless Format = U8 Range = [0,255]				

SFID_DP_DC1 is an extension of SFID_DP_DC0 to allow for more message types. They act as a single logical entity

The stateless aliases provide a means of SW controlling the coherency properties of an access. The property is ensured for that access only. Typically, SW will use 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	Coherency	Explanation
255	IA Coherent	Coherent within Gen and coherent within the entire IA cache memory hierarchy.
253	Non-Coherent	Coherent within Gen, same cache type.

Programming Note	
Context:	Message Descriptor
The constant, sampler and render cache are always non-coherent.	

Programming Note	
Context:	Message Descriptor
If SLICE_COMMON_CHICKEN_HDC0::Force Non-Coherent is enabled all SFID_DP_DC0 & SFID_DP_DC1 accesses are non-coherent.	

Message Header

This header applies to the following data port messages:

Data Port Message
OWord Block Read/Write
Unaligned OWord Block Read
OWord Dual Block Read/Write
DWord Scattered Read/Write
Byte Scattered Read/Write
Scratch Block Read/Write

The header definitions for the other data port messages is in the section for each message.

DWord	Bits	Description
M0.5	31:10	Immediate Buffer Base Address. Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), else this field is ignored. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:10]
	9:8	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.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:4	Ignored
	3:0	Per Thread Scratch Space Specifies the amount of scratch space allowed to be used by this thread for messages in which the Binding Table Index is 255 (stateless model), else this field is ignored. 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. The data port uses this to bounds check scratch space messages. Writes out of bounds are ignored. Reads out of bounds return 0. Format = U4 Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two.
M0.2	31:0	Global Offset. Specifies the global element offset into the buffer. For the Unaligned OWord messages, this offset is in units of Bytes but must be DWord-aligned (bits 1:0 MBZ). For the other OWord messages, this offset is in units of OWords. For the DWord messages, this offset is in units of DWords. For the Byte messages, this offset is in units of Bytes. Format = U32 Range = [0,FFFFFFFh] for Unaligned OWord messages. Range = [0,0FFFFFFFh] for other OWord messages. Range = [0,3FFFFFFFh] for DWord messages.

DWord	Bits	Description
		Range = [0,FFFFFFFFh] for Byte messages.
M0.1	31:0	Ignored
M0.0	31:0	Ignored

Write Commit Writeback Message

The writeback message is only sent on Data Port Write messages if the **Send Write Commit Message** bit in the message descriptor is set. The destination register is not modified. Write messages without the **Send Write Commit Message** bit set will not return anything to the thread (response length is 0 and destination register is null).

DWord	Bit	Description
W0.7:0		Reserved

OWord Block ReadWrite

This message takes one offset (Global Offset), and reads or writes 1, 2, 4, or 8 contiguous OWords starting at that offset.

Restrictions

Restriction
The only surface type allowed is SURFTYPE_BUFFER.
The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.
The surface cannot be tiled.
The surface base address must be OWord-aligned.
The Render Cache Read Write Mode field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.
The Stateless Render Cache Read-Write Mode field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model.

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.

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth registers (W1, W3 or M2, M4). For reads, any mask bit set within a group of four causes the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, are used as the single group of four with behavior following that in the preceding paragraph.

The above behavior enables a SIMD16 thread to use the 8-OWord form of this message to access two channels (red and green) of a single scratch register across 16 pixels. A second message would access the other two channels (blue and alpha). The execution mask is used to ensure that data associated with inactive pixels are not overwritten.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and do not modify memory.

Message Descriptor

Bits	Description						
13	Reserved: MBZ						
12	Ignored						
11	Ignored						
10:8	Block Size. Specifies the number of contiguous OWords to be read or written 000: 1 OWord, read into or written from the low 128 bits of the destination register. 001: 1 OWord, read into or written from the high 128 bits of the destination register. 010: 2 OWords 011: 4 OWords 100: 8 OWords All other encodings are reserved. <table border="1" data-bbox="219 688 1044 825"> <tr> <th colspan="2">Programming Note</th></tr> <tr> <td>Context:</td><td>Message Descriptor</td></tr> <tr> <td colspan="2">The 6 OWord block size is valid only with Data Port Constant Cache.</td></tr> </table>	Programming Note		Context:	Message Descriptor	The 6 OWord block size is valid only with Data Port Constant Cache .	
Programming Note							
Context:	Message Descriptor						
The 6 OWord block size is valid only with Data Port Constant Cache .							

Message Payload (Write)

For the write operation, the message payload consists of one, two, or four registers (not including the header) depending on the **Block Size** specified in the message. For the one-constant case, data is taken from either the high or low half of the payload register depending on the half selected in **Block Size**. In this case, the other half of the payload register is ignored.

The **Offset** referred to below is the **Global Offset** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bits	Description
M1.7:4	127:0	OWord[Offset + 1]. If the block size is 1, OWord to be written from the high 128 bits of the destination, OWord[Offset] will appear in this location.
M1.3:0	127:0	OWord[Offset]
M2.7:4	127:0	OWord[Offset+3]
M2.3:0	127:0	OWord[Offset+2]
M3.7:4	127:0	OWord[Offset+5]
M3.3:0	127:0	OWord[Offset+4]
M4.7:4	127:0	OWord[Offset+7]
M4.3:0	127:0	OWord[Offset+6]

Writeback Message (Read)

For the read operation, the writeback message consists of one, two, three, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Offset** referred to below is the **Global Offset** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bits	Description
W0.7:4	127:0	OWord[Offset + 1] . If the block size is 1, OWord to be loaded into the high 128 bits of the destination, OWord[Offset] will appear in this location.
W0.3:0	127:0	OWord[Offset]
W1.7:4	127:0	OWord[Offset+3]
W1.3:0	127:0	OWord[Offset+2]
W2.7:4	127:0	OWord[Offset+5]
W2.3:0	127:0	OWord[Offset+4]
W3.7:4	127:0	OWord[Offset+7]
W3.3:0	127:0	OWord[Offset+6]

Unaligned OWord Block Read

This message takes one DWord aligned offset (**Global Offset**), and reads 1, 2, 4, or 8 contiguous OWords starting at that offset. This message is identical to the OWord Block Read message except for the offset alignment. For read/write cache, only the read path supports this unaligned OWord Block access.

Restrictions

Restriction
The only surface type allowed is SURFTYPE_BUFFER.
The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.
The surface cannot be tiled.
The surface base address must be OWord-aligned.
The Render Cache Read Write Mode field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.
The Stateless Render Cache Read-Write Mode field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model.

Applications: Reads with an offset that is not aligned with data size, such as row store usage in media.

Execution Mask. The execution mask is ignored by this message.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0.

Message Descriptor

Bits	Description
13	Ignored
12:11	Ignored
10:8	Block Size. Specifies the number of contiguous OWords to be read. 000: 1 OWord, read into the low 128 bits of the destination register. 001: 1 OWord, read into the high 128 bits of the destination register. 010: 2 OWords. 011: 4 OWords. 100: 8 OWords. All other encodings are reserved.

Writeback Message (Read)

For the read operation, the writeback message consists of one, two, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Global Offset** is in units of **Bytes**, aligned to **DWord** (two LSBs set to zero). The **OWordX** array in units of OWord starts at Global Offset.

DWord	Bit	Description
W0.7:4	127:0	OWord1 = *(\&OWord0 + 1) . If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord0 will appear in this location
W0.3:0	127:0	OWord0 = Buffer[Global Offset]
W1.7:4	127:0	OWord3 = *(\&OWord2 + 1)
W1.3:0	127:0	OWord2 = *(\&OWord1 + 1)
W2.7:4	127:0	OWord5 = *(\&OWord4 + 1)
W2.3:0	127:0	OWord4 = *(\&OWord3 + 1)
W3.7:4	127:0	OWord7 = *(\&OWord6 + 1)
W3.3:0	127:0	OWord6 = *(\&OWord5 + 1)

OWord Dual Block ReadWrite

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.

The message header is no longer required for the *OWord Dual Block Read/Write* messages if sent to the data cache data port. If header is not sent, the **Global Offset** field is assumed to be zero. The header is required, however, if the binding table index is 255 (stateless model), as the **Immediate Buffer Base Address** field is required.

Programming Restriction: Writes to overlapping addresses have undefined write ordering.

Restrictions

Restriction
The only surface type allowed is SURFTYPE_BUFFER.
The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.
The surface cannot be tiled.
The surface base address must be OWord-aligned.
The Render Cache Read Write Mode field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.
The Stateless Render Cache Read-Write Mode field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model.

Applications:

- SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access).
- SIMD4x2 scratch space reads/writes where the indices are different.

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the GRF registers returned for read, or each of the write registers sent. For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and do not modify memory contents.

Message Descriptor

Bits	Description
13	Reserved: MBZ
12:10	Ignored
9:8	Block Size. Specifies the number of OWords in each block to be read or written: 00: 1 OWord 10: 4 OWords All other encodings are reserved.

Message Payload

DWord	Bits	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored
M1.4	31:0	Block Offset 1. Specifies the OWord offset of OWord Block 1 into the surface. Format = U32 Range = [0,0FFFFFFh]
M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	Block Offset 0

Additional Message Payload (Write)

For the write operation, the message payload consists of one or four registers (not including the header or the first part of the payload) depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bit	Description
M2.7:4	127:0	OWord[Offset1]
M2.3:0	127:0	OWord[Offset0]
M3.7:4	127:0	OWord[Offset1+1]
M3.3:0	127:0	OWord[Offset0+1]
M4.7:4	127:0	OWord[Offset1+2]
M4.3:0	127:0	OWord[Offset0+2]
M4.7:4	127:0	OWord[Offset1+3]
M4.3:0	127:0	OWord[Offset0+3]

Writeback Message (Read)

For the read operation, the writeback message consists of one or four registers depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bits	Description
W0.7:4	127:0	OWord[Offset1]
W0.3:0	127:0	OWord[Offset0]
W1.7:4	127:0	OWord[Offset1+1]
W1.3:0	127:0	OWord[Offset0+1]
W2.7:4	127:0	OWord[Offset1+2]
W2.3:0	127:0	OWord[Offset0+2]
W3.7:4	127:0	OWord[Offset1+3]
W3.3:0	127:0	OWord[Offset0+3]

Media Block ReadWrite

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.

Restrictions

Restriction
The only surface type allowed is non-arrayed, non-mipmapped SURFYPE_2D. Because of this, the stateless surface model is not supported with this message.
Accesses are allowed to SURFYPE_NULL, reads return 0 and writes are ignored.
The surface format is used to determine the pixel structure for boundary clamp; the raw data from the surface is returned to the thread without any format conversion nor filtering operation.
The target cache cannot be the data cache.
The surface base address must be 32-byte aligned.
When a surface is XMajor tiled, (tilewalk field in the surface state is set to TILEWALK_XMAJOR), a memory area mapped through the Render Cache cannot be read and/or written in mixed frame and field modes. For example, if a memory location is first written with a zero Vertical Line Stride (frame mode), and later on (without render cache flush) read back using Vertical Line Stride of one (field mode), the read data stored in the GRF are uncertain.
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.
For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. DWord-aligned).

Restriction
The write form of this message has the additional restriction that both X Offset and Block Width must be DWord-aligned.
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.

Applications: Block reads/writes for media.

Execution Mask. The execution mask on the *send* instruction for this type of message is ignored. The data that is read or written is determined completely by the block parameters.

Out-of-Bounds Accesses. Reads outside of the surface results in the address being clamped to the nearest edge of the surface and the pixel in the position being returned. Writes outside of the surface are dropped and will not modify memory contents.

Determining the boundary pixel value depends on the surface format. Surface format definitions can be found in the Surface Formats Section of the Sampling Engine Chapter.

For a surface with 8bpp pixels, the boundary byte is replicated. For example, for a boundary DWord B0B1B2B3, to replicate the left boundary byte pixel, the out of bound DWords have the format B0B0B0B0, and the format for the right boundary is B3B3B3B3.

This rule applies to all surface formats with BPE of 8. As the data port does not perform format conversion, the most likely used surface formats are R8_UINT and R8_SINT.

For any other surfaces with 16bpp pixels, boundary pixel replication is on words. For example, for a boundary dword B0B1B2B3, to replicate the left boundary word pixel, the out of bound DWords have the format B0B1B0B1, and the format for the right boundary is B2B3B2B3.

This rule applies to all surface formats with BPE of 16. As the data port does not perform format conversion, only the formats with integer data types may be useful in practice.

For special surfaces with 16bpp pixels YUV422 packed format, there are two basic cases depending on the Y location: YUYV (surface format YCRCB_NORMAL) and UYVY (surface format YCRCB_SWAPY). Boundary handling for YVYU (surface format YCRCB_SWAPUV) is the same as that for YUYV. Similarly, boundary handling for VYUY (surface format YCRCB_SWAPUVY) is the same as that for UYVY. Note that these four surface formats have 16bpp pixels, even though the BPE fields are set to zero according to the table in the Surface Formats Section.

For a boundary DWord Y0U0Y1V0, to replicate the left boundary, we get Y0U0**Y0**V0, and to replicate the right boundary, we get **Y1**U0Y1V0.

For a boundary DWord U0Y0V0Y1, to replicate the left boundary, we get U0Y0V0**Y0**, and to replicate the right boundary, we get U0**Y1**V0Y1.

For a surface with 32bpp pixels, the boundary DWord pixel is replicated.

This rule applies to all surface formats with BPE of 32. As the data port does not perform format conversion, some of the formats may not be useful in practice.

Hardware behavior for any other surface types is undefined.

NV12 surface state: The width of the surface should be always multiples of 4 pixels. For 16bpp input message (422 8-bit) the width will always need to be in multiples of 8 bytes and for 32bpp input message (422 16-bit or 444 8-bit) the width should be in multiples of 16 bytes. Height should be in multiples of 2 pixels high. (Presently the MFX restriction is that width should be in multiples of 2 pixels.)

The y-offset of the media block write from the EU should always be even.

The x-offset of the media block write from the EU should be in multiples of 4 pixels.

The media block DWord write can have only the following combinations (for IECP when NV12 output format is used):

- 8 pixels wide for 422 8-bit mode
- 4 pixels wide for 422 8-bit mode
- 4 pixels wide for 422 16-bit
- 4 pixels wide for 444 8-bit
- 444 16-bit input format cannot be supported when the output format is NV12 (SW should not use this combination).
- It has to be in multiples of 2 pixels high for all above modes.

If 444-format is used then we use only the pixel_0 UV values of the 2x2 pixel and the rest are dropped and in case of 422-format the top UV values are used and the bottom UV values are dropped if the output format is NV12 format.

Assuming IECP messages will always have vertical stride = 0 (since this is only for pre-processing before the encoder).

Message Descriptor

Bit	Description															
13	Reserved: MBZ															
12	Reserved: MBZ															
11	Reserved: MBZ															
10	<p>Vertical Line Stride Override</p> <p>Specifies whether the Vertical Line Stride and Vertical Line Stride Offset fields in the surface state should be replaced by bits 9 and 8 below.</p> <p>If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:</p> <table><tr><th>Vertical Line Stride (in surface state)</th><th>Override Vertical Line Stride</th><th>Derived 1-based Surface Height (As a function of the 0-based Height in Surface State)</th></tr><tr><td>0</td><td>0</td><td>Height + 1 (Normal)</td></tr><tr><td>0</td><td>1</td><td>(Height + 1) / 2 Restriction: (Height + 1) must be an even number.</td></tr><tr><td>1</td><td>0</td><td>(Height + 1) * 2</td></tr><tr><td>1</td><td>1</td><td>Height + 1 (Normal)</td></tr></table> <p>For example, for a 720x480 standard resolution video buffer, if Vertical Line Stride in surface state is 0, i.e. a frame, Height (of the frame) should be 479. When accessing the bottom field of this frame video buffer, if both Override Vertical Line Stride and Override Vertical Line Stride Offset are set to 1, then the derived surface height (of the field) is 240 ((Height + 1) / 2). In contrast, if Vertical Line Stride in surface state is 1 and Vertical Line Stride Offset in surface state is 0, the surface state represents the top field of the video buffer. In this case, Height (of the top field) should be programmed as 239. Accessing the bottom video field uses the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset of 0) results in a derived surface height of 480 ((Height + 1) * 2).</p> <p>0: Use parameters in the surface state and ignore bits 9:8.</p> <p>1: Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset.</p>	Vertical Line Stride (in surface state)	Override Vertical Line Stride	Derived 1-based Surface Height (As a function of the 0-based Height in Surface State)	0	0	Height + 1 (Normal)	0	1	(Height + 1) / 2 Restriction: (Height + 1) must be an even number.	1	0	(Height + 1) * 2	1	1	Height + 1 (Normal)
Vertical Line Stride (in surface state)	Override Vertical Line Stride	Derived 1-based Surface Height (As a function of the 0-based Height in Surface State)														
0	0	Height + 1 (Normal)														
0	1	(Height + 1) / 2 Restriction: (Height + 1) must be an even number.														
1	0	(Height + 1) * 2														
1	1	Height + 1 (Normal)														
9	<p>Override Vertical Line Stride</p> <p>Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures.</p> <p>Format = U1 in lines to skip between logically adjacent lines.</p>															
8	<p>Override Vertical Line Stride Offset</p> <p>Specifies the offset of the initial line from the beginning of the buffer. Ignored when Override VerticalLine Stride is 0.</p> <p>Format = U1 in lines of initial offset (when Vertical Line Stride == 1).</p>															

Message Header

DWord	Bits	Description																																																																
M0.5	31:8	Ignored																																																																
	7:0	FFTID . 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.																																																																
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)																																																																
M0.3	31:0	<p>Pixel Mask. One bit per pixel (each pixel being a DWord) indicating which pixels are to be written. This field is ignored by the read message, all pixels are always returned. This field is used only if the Message Mode field is set to PIXEL_MASK. The bits in this mask correspond to the pixels (DWords) as follows:</p> <table><tr><td>0</td><td>1</td><td>4</td><td>5</td><td>16</td><td>17</td><td>20</td><td>21</td></tr><tr><td>2</td><td>3</td><td>6</td><td>7</td><td>18</td><td>19</td><td>22</td><td>23</td></tr><tr><td>8</td><td>9</td><td>12</td><td>13</td><td>24</td><td>25</td><td>28</td><td>29</td></tr><tr><td>10</td><td>11</td><td>14</td><td>15</td><td>26</td><td>27</td><td>30</td><td>31</td></tr></table> <p>Byte Mask One bit per byte indicating which bytes are to be written. This field is ignored by the read message; all bytes are always returned. This field is used only if the Message Mode is BYTE_MASK. The Byte Mask is a horizontal mask applied to each row of output.</p> <table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr><tr><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td></tr><tr><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr></table>	0	1	4	5	16	17	20	21	2	3	6	7	18	19	22	23	8	9	12	13	24	25	28	29	10	11	14	15	26	27	30	31	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
0	1	4	5	16	17	20	21																																																											
2	3	6	7	18	19	22	23																																																											
8	9	12	13	24	25	28	29																																																											
10	11	14	15	26	27	30	31																																																											
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																																																											
M0.2	31:30	<p>Message Mode. This field selects the mode of this message as follows:</p> <p>00: NORMAL. The Block Height and Block Width fields are set in M0.2. The Pixel Mask is not explicitly set but behaves as if it is set to all ones.</p> <p>01: PIXEL_MASK: The Pixel Mask field is set in M0.3. The Block Height and Block Width are not explicitly set but behave as if they are set to 4 rows and 32 bytes, respectively.</p> <p>10: BYTE_MASK: The Block Height and Block Width fields are set in M0.2. The Byte Mask in M0.3 is used to qualify which bytes are written.</p> <p>11: Reserved.</p> <p>For the <i>Sampler Cache Data Port</i>, this field is also ignored, behaving as if always set to NORMAL.</p>																																																																
	29	Ignored																																																																
	28:24	<p>Sub-Register Offset. This field provides the sub-register offset in unit of byte of a media block read message. This field is ignored (reserved) for a media block write message.</p> <p>Programming Notes:</p> <p>Sub-Register Offset must be aligned to BasePitch (therefore will be a multiple of DWords as well).</p>																																																																

DWord	Bits	Description												
		<p>When Register Pitch Control = 0, Sub-Register Offset must align to BasePitch*Block Height and the output fits in a single GRF register.</p> <p>In general (and specifically when Sub-Register Offset is greater than 0), when the resulting data cross a GRF register boundary, the data must be placed symmetrically between GRF registers.</p> <p>Sub-Register Offset and Register Pitch Control allow software to assemble multiple media block reads directly into a shared GRF register set. For example, if both are set to zero, the read data are written to GRF registers, aligning 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. If Register Pitch Control is non-zero, 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 media block data line interleaved.</p> <p>This field must be zero for Render Cache Data Port.</p> <p>Format = U5</p> <p>Range = [0, 28] (Only a multiple of BasePitch, including 0, is valid.)</p>												
	21:16	<p>Block Height. Height in rows of block being accessed.</p> <p>Programming Note: The Block Height is restricted to the following maximum values depending on the Block Width:</p> <table><tr><th>Block Width (bytes)</th><th>Maximum Block Height (rows)</th></tr><tr><td>1-4</td><td>64</td></tr><tr><td>5-8</td><td>32</td></tr><tr><td>9-16</td><td>16</td></tr><tr><td>17-32</td><td>8</td></tr><tr><td>33-64</td><td>4</td></tr></table> <p>Programming Note: Block width > 32 bytes is allowed only for media block write messages and only for linear and Tile X surfaces.</p> <p>Format = U6</p> <p>Range = [0,63] representing 1 to 64 rows</p>	Block Width (bytes)	Maximum Block Height (rows)	1-4	64	5-8	32	9-16	16	17-32	8	33-64	4
Block Width (bytes)	Maximum Block Height (rows)													
1-4	64													
5-8	32													
9-16	16													
17-32	8													
33-64	4													
	15:10	Ignored												
	9:8	<p>Register Pitch Control. This field controls the register pitch for a media block read message.</p> <p>Register Pitch Control is only allowed to be non-zero if Block Width is a multiple of DWords. The effective register pitch must be less than or equal to 32 bytes (to fit in a single GRF register).</p> <p>Defining BasePitch as the next power-of-2 that is greater than or equal to the Block Width, Register Pitch Control set the register pitch in terms of BasePitch as the following.</p> <p>Range = [0,3] representing 1 to 4 BasePitch</p> <p>Programming Notes:</p>												

DWord	Bits	Description
		<p>This field must be zero for Render Cache Data Port.</p> <p>This field is reserved (must be zero)) for a media block write message.</p> <p>Programming Restriction: This field must be zero.</p>
	7:6	Ignored
	5:0	<p>Block Width. Width in bytes of the block being accessed.</p> <p>Programming Note: Must be DWord-aligned for the write form of the message.</p> <p>Format = U6</p> <p>Range = [0,63] representing 1 to 64 bytes</p>
M0.1	31:0	<p>Y offset. The Y offset of the upper left corner of the block into the surface.</p> <p>Format = S31</p> <p>Programming Note: If Message Mode is set to PIXEL_MASK, this field must be a multiple of 4.</p>
M0.0	31:0	<p>X offset. The X offset of the upper left corner of the block into the surface.</p> <p>Must be DWord-aligned (Bits 1:0 MBZ) for the write form of the message.</p> <p>The X offset field defines the offset in the input message block. This may differ from the offset in the surface if Color Processing is enabled due to format conversion.</p> <p>Format = S31</p> <p>Programming Note: If Message Mode is set to PIXEL_MASK, this field must be a multiple of 32.</p>

The legal combinations of block width, pitch control, sub-register offset, and block height are given below:

Block Height for given block width, pitch control, subreg offsets									
block width	pitch control	sub-register offsets							
		0	1	2	3	4	5	6	7
1-4	00	1-64	1	1	1	1	1	1	1
	01	1-64	1-64	illegal	illegal	1-2	1-2	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-64	1-64	1-64	1-64	illegal	illegal	illegal	illegal
5-8	00	1-32	illegal	1	illegal	1	illegal	1	illegal
	01	1-32	illegal	1-32	illegal	illegal	illegal	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-32	illegal	1-32	illegal	1-32	illegal	1-32	illegal
9-16	00	1-16	illegal	illegal	illegal	1	illegal	illegal	illegal
	01	1-16	illegal	illegal	illegal	1-16	illegal	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-16	illegal	illegal	illegal	1-16	illegal	illegal	illegal
17-32	00	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	01	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal

Message Payload (Write)

DWord	Bit	Description
M1:n		Write Data. The format of the write data 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 .

Writeback Message (Read)

DWord	Bits	Description
W0:n	31:0	Read Data. The format of the read data depends on the Block Height , Block Width , Register Pitch Control , and Sub-Register Offset . The data is aligned to the Sub-Register Offset of the first register, and the register pitch is set to one or more BasePatch .

DWord Scattered ReadWrite

This message takes a set of offsets, and reads or writes 8 or 16 scattered DWords starting at each offset. The Global Offset is added to each of the specific offsets.

Restrictions

Restriction
The only surface type allowed is SURFTYPE_BUFFER.
The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.
The surface cannot be tiled.
The surface base address must be DWord-aligned.
Writes to overlapping addresses have undefined write ordering.
For read messages with X/Y offsets that are outside the bounds of the surface, the address is clamped to the nearest edge of the surface. For write messages with X/Y offsets that are outside the bounds of the surface, the behavior is undefined.
The Render Cache Read Write Mode field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.
The Stateless Render Cache Read-Write Mode field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model.
Hardware does check for and optimize for cases where offsets are equal or contiguous, however for optimal performance in some of these cases a different message may provide higher performance.
The message header is no longer required for the <i>OWord DWord Scattered Read/Write</i> messages if sent to the data cache data port. If header is not sent, the Global Offset field is assumed to be zero. The header is required, however, if the binding table index is 255 (stateless model), as the Immediate Buffer Base Address field is required.

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

Execution Mask. Depending on the block size, either the low 8 bits or all 16 bits of the execution mask are used to determine which DWords are read into the destination GRF register (for read), or which DWords are written to the surface (for write).

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

Message Descriptor

Bits	Description
13	Reserved: MBZ
12	Ignored
11:10	Ignored
9:8	Block Size. Specifies the number of DWords read or written: 10: 8 DWords 11: 16 DWords All other encodings are reserved.

Message Payload

DWord	Bits	Description
M1.7	31:0	Offset 7. Specifies the DWord offset of DWord 7 into the surface. Format = U32 Range = [0,3FFFFFFh]
M1.6	31:0	Offset 6
M1.5	31:0	Offset 5
M1.4	31:0	Offset 4
M1.3	31:0	Offset 3
M1.2	31:0	Offset 2
M1.1	31:0	Offset 1
M1.0	31:0	Offset 0
M2.7	31:0	Offset 15. This message register is included only if the block size is 16 DWords.
M2.6	31:0	Offset 14
M2.5	31:0	Offset 13
M2.4	31:0	Offset 12
M2.3	31:0	Offset 11
M2.2	31:0	Offset 10
M2.1	31:0	Offset 9
M2.0	31:0	Offset 8

Additional Message Payload (Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M3.7	31:0	DWord[Offset7]
M3.6	31:0	DWord[Offset6]
M3.5	31:0	DWord[Offset5]
M3.4	31:0	DWord[Offset4]
M3.3	31:0	DWord[Offset3]
M3.2	31:0	DWord[Offset2]
M3.1	31:0	DWord[Offset1]
M3.0	31:0	DWord[Offset0]
M4.7	31:0	DWord[Offset15] . This message register is included only if the block size is 16 DWords
M4.6	31:0	DWord[Offset14]
M4.5	31:0	DWord[Offset13]
M4.4	31:0	DWord[Offset12]
M4.3	31:0	DWord[Offset11]
M4.2	31:0	DWord[Offset10]
M4.1	31:0	DWord[Offset9]
M4.0	31:0	DWord[Offset8]

Writeback Message (Read)

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **DWord** array index is also in units of DWords.

DWord	Bits	Description
W0.7	31:0	DWord[Offset7]
W0.6	31:0	DWord[Offset6]
W0.5	31:0	DWord[Offset5]
W0.4	31:0	DWord[Offset4]
W0.3	31:0	DWord[Offset3]
W0.2	31:0	DWord[Offset2]
W0.1	31:0	DWord[Offset1]
W0.0	31:0	DWord[Offset0]
W1.7	31:0	DWord[Offset15]. This writeback message register is included only if the block size is 16 DWords.
W1.6	31:0	DWord[Offset14]
W1.5	31:0	DWord[Offset13]
W1.4	31:0	DWord[Offset12]
W1.3	31:0	DWord[Offset11]
W1.2	31:0	DWord[Offset10]
W1.1	31:0	DWord[Offset9]
W1.0	31:0	DWord[Offset8]

Message Descriptor

Bit	Description
12	Two-Source Message. When this bit is set, there are two data-phases for two sources. Two-source message is used only for opcode "0111" and for all other opcodes this bit must be 0. When this bit is 0, M3 is not sent to the data-port.
11:8	Atomic Operation Code: (Please refer to the table below) Unsupported opcodes: 1101, 1110, 1111

Opcode	Operation	Return Value
0000	ADD: new = old + src0	Old value
0001	SUB: new = old – src0	Old value
0010	INC : new = old+1	Old value
0011	DEC: new = old-1	Old value
0100	MIN: new = min(old, src0)	Old value
0101	MAX: new = max(old, src0)	Old value
0110	XCHG: new = src0	Old value
0111	CMPXCHG : new = (old==src1) ? src0 : old	Old value
1000	AND: new = old & src0	Old value
1001	OR: new = old src0	Old value
1010	XOR: new = old ^ src0	Old value
1011	MIN_SINT: new = min(old, src0)	Old value(signed)
1100	MAX_SINT: new = max(old, src0)	Old value(signed)
1101-1111		Old value

Message Payload

DWord	Bit	Description
M1.7	31:0	Offset 7. Specifies the DWord offset of DWord 7 into the surface. Format = U32 Range = [0,3FFFFFFh]
M1.6	31:0	Offset 6
M1.5	31:0	Offset 5
M1.4	31:0	Offset 4
M1.3	31:0	Offset 3
M1.2	31:0	Offset 2
M1.1	31:0	Offset 1
M1.0	31:0	Offset 0

Source Payload

Either one or two additional registers (depending on **Two-Source Message**) of source payload contain the data to be used as source.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M2.7	31:0	DWord[Offset7] Src0
M2.6	31:0	DWord[Offset6] Src0
M2.5	31:0	DWord[Offset5] Src0
M2.4	31:0	DWord[Offset4] Src0
M2.3	31:0	DWord[Offset3] Src0
M2.2	31:0	DWord[Offset2] Src0
M2.1	31:0	DWord[Offset1] Src0
M2.0	31:0	DWord[Offset0] Src0
M3.7	31:0	DWord[Offset7] Src1
M3.6	31:0	DWord[Offset6] Src1
M3.5	31:0	DWord[Offset5] Src1
M3.4	31:0	DWord[Offset4] Src1
M3.3	31:0	DWord[Offset3] Src1
M3.2	31:0	DWord[Offset2] Src1
M3.1	31:0	DWord[Offset1] Src1
M3.0	31:0	DWord[Offset0] Src1

Writeback Message

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **Offset_n** referred to below is the **Global Offset** added to the corresponding **Offset *n*** and is in units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
W0.7	31:0	DWord[Offset7]
W0.6	31:0	DWord[Offset6]
W0.5	31:0	DWord[Offset5]
W0.4	31:0	DWord[Offset4]
W0.3	31:0	DWord[Offset3]
W0.2	31:0	DWord[Offset2]
W0.1	31:0	DWord[Offset1]
W0.0	31:0	DWord[Offset0]

Byte Scattered ReadWrite

These messages are supported on BDW only.

These messages take a set of offsets, and read or write 8 or 16 scattered and possibly misaligned bytes, words, or DWords starting at each offset. The **Global Offset** from the message header is added to each of the specific offsets.

Restrictions

Restriction
The only surface type allowed is SURFTYPE_BUFFER.
The surface format is ignored; data is returned from the buffer to the GRF without format conversion.
The surface cannot be tiled.
The surface base address must be DWord-aligned.
Writes to overlapping addresses have undefined write ordering.
The surface is treated as a 1-dimensional surface. The element size (pitch) times the number of elements is used to determine the size of the buffer for out-of-bounds checking if using the surface state model. Out of bounds checking is done at DWord granularity; if any part of the DWord is out-of-bounds then the whole DWord is considered out-of-bounds.

Applications: Byte aligned buffer accesses in GPGPU programs.

Execution Mask. Depending on the block size, either the low 8 bits or all 16 bits of the execution mask are used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write).

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

Message Descriptor

Bit	Description
13:12	Ignored
11:10	Data Size. Specifies the data size for each slot. 0: 1 byte 1: 2 bytes 2: 4 bytes 3: Reserved
9	Ignored
8	SIMD Mode. Specifies the SIMD mode of the message (number of slots processed). 0: SIMD8 1: SIMD16

Message Payload

DWord	Bits	Description
M1.7	31:0	Offset 7. Specifies the byte offset of DWord 7 into the surface. Format = U32 Range = [0,FFFFFFFFh]
M1.6	31:0	Offset 6
M1.5	31:0	Offset 5
M1.4	31:0	Offset 4
M1.3	31:0	Offset 3
M1.2	31:0	Offset 2
M1.1	31:0	Offset 1
M1.0	31:0	Offset 0
M2.7	31:0	Offset 15. This message register is included only if the SIMD Mode is SIMD16.
M2.6	31:0	Offset 14
M2.5	31:0	Offset 13
M2.4	31:0	Offset 12
M2.3	31:0	Offset 11
M2.2	31:0	Offset 10
M2.1	31:0	Offset 9
M2.0	31:0	Offset 8

Additional Message Payload (Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offset_n** referred to below is the **Global Offset** added to the corresponding **Offset *n*** and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field. The upper bits are ignored for 1 byte and 2 byte **Data Size**.

DWord	Bit	Description
M3.7	31:0	Data[Offset7]
M3.6	31:0	Data[Offset6]
M3.5	31:0	Data[Offset5]
M3.4	31:0	Data[Offset4]
M3.3	31:0	Data[Offset3]
M3.2	31:0	Data[Offset2]
M3.1	31:0	Data[Offset1]
M3.0	31:0	Data[Offset0]
M4.7	31:0	Data[Offset15] . This message register is included only if the SIMD Mode is SIMD16.
M4.6	31:0	Data[Offset14]
M4.5	31:0	Data[Offset13]
M4.4	31:0	Data[Offset12]
M4.3	31:0	Data[Offset11]
M4.2	31:0	Data[Offset10]
M4.1	31:0	Data[Offset9]
M4.0	31:0	Data[Offset8]

Writeback Message (Read)

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **Offset_n** referred to below is the **Global Offset** added to the corresponding **Offset *n*** and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field and only the requested bytes are written to the GRF.

DWord	Bit	Description
W0.7	31:0	Data[Offset7]
W0.6	31:0	Data[Offset6]
W0.5	31:0	Data[Offset5]
W0.4	31:0	Data[Offset4]
W0.3	31:0	Data[Offset3]
W0.2	31:0	Data[Offset2]
W0.1	31:0	Data[Offset1]
W0.0	31:0	Data[Offset0]
W1.7	31:0	Data[Offset15] . This message register is included only if the SIMD Mode is SIMD16.
W1.6	31:0	Data[Offset14]
W1.5	31:0	Data[Offset13]
W1.4	31:0	Data[Offset12]
W1.3	31:0	Data[Offset11]
W1.2	31:0	Data[Offset10]
W1.1	31:0	Data[Offset9]
W1.0	31:0	Data[Offset8]

TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation

Six data port messages (Typed Surface Read, Typed Surface Write, Typed Atomic Operation, Untyped Surface Read, Untyped Surface Write, and Untyped Atomic Operation) allow direct read/write accesses to surfaces. These messages support three major categories of surfaces:

Typed surfaces. These surfaces are of type SURFTYPE_1D, 2D, 3D, or BUFFER and have a supported surface format other than RAW.

A typed surface uses U, V, R, and LOD address parameters (the number of parameters used depends on the surface type), and performs conversion of type to/from the selected surface format as follows:

- Surface formats with UINT require the message data in U32 format.
- Surface formats with SINT require the message data in S32 format.
- All other surface formats require the message data in FLOAT32 format.

The untyped surface categories, both of which use the RAW surface format, perform no type conversion. A raw buffer uses just the U address parameter, which specifies the byte offset into the surface, which must be a multiple of 4. A structured buffer uses the U address parameter as an array index and the V address parameter as a byte offset into the array element (which also must be a multiple of 4).

For both raw and structured buffers, up to 4 DWords are accessed beginning at the byte address determined. These 4 dwords correspond to the red, green, blue, and alpha channels in that order with red mapping to the lowest order DWord. The atomic operation messages only access the first DWord (corresponding to the red channel for typed messages).

The atomic operation messages cause atomic read-modify-write operations on the “destination” location addressed. In the table below, the new value of the destination (new_dst) is computed as indicated based on the old value of the destination (old_dst) and up to two sources included in the message (src0 and src1). Optionally, a value can be returned by the message (ret).

The atomic operations guarantee that the read and the write are performed atomically, meaning 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.

The following atomic operations are available, along with the specific operation performed for each and the return value:

Atomic Operation	new_dst	ret
AOP_AND	old_dst & src0	old_dst
AOP_OR	old_dst src0	old_dst
AOP_XOR	old_dst ^ src0	old_dst
AOP_MOV	src0	old_dst
AOP_INC	old_dst + 1	old_dst
AOP_DEC	old_dst – 1	old_dst
AOP_ADD	old_dst + src0	old_dst

Atomic Operation	new_dst	ret
AOP_SUB	old_dst – src0	old_dst
AOP_REVSUB	src0 – old_dst	old_dst
AOP_IMAX	imax(old_dst, src0)	old_dst
AOP_IMIN	imin(old_dst, src0)	old_dst
AOP_UMAX	umax(old_dst, src0)	old_dst
AOP_UMIN	umin(old_dst, src0)	old_dst
AOP_CMPWR	(src0 == old_dst) ? src1 : old_dst	old_dst
AOP_PREDEC	old_dst – 1	new_dst
AOP_CMPWR8B	(src08B == old_dst8B) ? src18B : old_dst8B	old_dst8B

Programming Note

Context: TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
src08B is 8 bytes, src18B is 8 bytes, and old_dst8B is 8 bytes in length.

Programming Note

Context: TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
AOP_CMPWR8B is not supported for SLM.

Programming Note

Context: TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
AOP_CMPWR8B addresses must be QWord-aligned.

Programming Note

Context: TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation

imax/imin assume operands are signed 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.

Programming Note

Context: TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation

For untyped messages, the **Tile Mode** must be LINEAR.

For untyped messages, the **Surface Format** must be RAW and the **Surface Type** must be SURFTYPE_BUFFER or SURFTYPE_STRBUF.

For typed messages, the **Surface Type** must be SURFTYPE_1D, 2D, 3D, or BUFFER.

Surface Format for Typed Surface Reads

Surface Format Name
R16G16B16A16_UINT
R8G8B8A8_UINT
R16G16_UINT
R32_SINT
R32_UINT
R32_FLOAT
R8G8_UINT
R16_UINT
R8_UINT

Surface Format for Typed Surface Writes

Surface Format Name
R32G32B32A32_FLOAT
R32G32B32A32_SINT
R32G32B32A32_UINT
R16G16B16A16_UNORM
R16G16B16A16_SNORM
R16G16B16A16_SINT
R16G16B16A16_UINT
R16G16B16A16_FLOAT
R32G32_FLOAT
R32G32_SINT
R32G32_UINT
B8G8R8A8_UNORM
R10G10B10A2_UNORM
R10G10B10A2_UINT
R8G8B8A8_UNORM
R8G8B8A8_SNORM
R8G8B8A8_SINT
R8G8B8A8_UINT
R16G16_UNORM
R16G16_SNORM
R16G16_SINT
R16G16_UINT

Surface Format Name
R16G16_FLOAT
B10G10R10A2_UNORM
R11G11B10_FLOAT
R32_SINT
R32_UINT
R32_FLOAT
B5G6R5_UNORM
B5G5R5A1_UNORM
B4G4R4A4_UNORM
R8G8_UNORM
R8G8_SNORM
R8G8_SINT
R8G8_UINT
R16_UNORM
R16_SNORM
R16_SINT
R16_UINT
R16_FLOAT
B5G5R5X1_UNORM
R8_UNORM
R8_SNORM
R8_SINT
R8_UINT
A8_UNORM

General Restrictions

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.

The **Surface Format** for typed atomic operations must be R32_UINT or R32_SINT.

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).

For untyped messages accessing SURFTYPE_STRBUF, the V address (byte offset) must be DWord-aligned (low 2 bits must be zero).

For untyped messages accessing SURFTYPE_BUFFER, the U address (byte offset) must be DWord-aligned (low 2 bits must be zero).

Typed messages only support SIMD8.

Project-Specific Restrictions

Programming Note	
Context:	TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
The stateless model support is limited to untyped messages. Furthermore, they are treated as SURFTYPE_BUFFER and Surface Format of RAW. The bounds checking for the stateless message is 4GB overflow and < General State upper bound.	

Programming Note	
Context:	TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
Tile W surfaces must be of format R8_UINT and only support SIMD8. Furthermore, only the RED channel can be enabled.	

Execution Mask

SIMD16: The 16 bits of the execution mask are ANDed with the 16 bits of the **Pixel/Sample Mask** from the message header and the resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the execution mask is used.

SIMD8: The low 8 bits of the execution mask are ANDed with 8 bits of the **Pixel/Sample Mask** from the message header. For the typed messages, the **Slot Group** in the message descriptor selects either the low or high 8 bits. For the untyped messages, the low 8 bits are always selected. The resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the low 8 bits of the execution mask are used.

SIMD4x2: Each group of 4 bits within the low 8 bits of the execution mask are ORed together to create two bits that are used to determine which slots are read into the destination GRF register.

Out-of-Bounds Accesses: Reads to areas outside of the surface return 0, except for the *Typed Surface Read* message that returns 1 in the alpha channel and 0 in the other channels. Writes to areas outside of the surface are dropped and will not modify memory contents.

Programming Note	
Context:	TypedUntyped Surface ReadWrite and TypedUntyped Atomic Operation
Writes to overlapping addresses have undefined write ordering.	

SIMD Mode, Surface Category, and Message Type Combinations Supported

SIMD Mode	Surface Category	Message Type
SIMD16	Untyped	Read
SIMD16	Untyped	Write
SIMD16	Untyped	Atomic
SIMD8	Untyped	Read
SIMD8	Untyped	Write
SIMD8	Untyped	Atomic
SIMD8	Typed	Read
SIMD8	Typed	Write
SIMD8	Typed	Atomic
SIMD4x2	Untyped	Read
SIMD4x2	Untyped	Write
SIMD4x2	Untyped	Atomic
SIMD4x2	Typed	Read
SIMD4x2	Typed	Write
SIMD4x2	Typed	Atomic

The following table indicates the hardware interpretation of each input parameter based on surface type. Parameters with blank entries are ignored by hardware if delivered.

Surface Type	"Surface Array" Field in SURFACE_STATE	U Address	V Address	R Address	LOD
SURFTYPE_1D	disabled	X pixel address			LOD
	enabled	X pixel address	array index		LOD
SURFTYPE_2D	disabled	X pixel address	Y pixel address		LOD
	enabled	X pixel address	Y pixel address	array index	LOD
SURFTYPE_3D	disabled	X pixel address	Y pixel address	Z pixel address	LOD
SURFTYPE_BUFFER	disabled	buffer index			
SURFTYPE_STRBUF	disabled	buffer index	byte offset		

Typed Surface ReadWrite Message Descriptor

Bit	Description
13	<p>Slot Group</p> <p>This field controls which 8 bits of Pixel/Sample Mask in the message header are ANDed with the execution mask to determine which slots are accessed. This field is ignored if the header is not present.</p> <p>Format = U1</p> <p>0: Use low 8 slots</p> <p>1: Use high 8 slots</p>
12	Ignored
11	<p>Alpha Channel Mask</p> <p>For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface.</p> <p>0: Alpha channel included</p> <p>1: Alpha channel not included</p> <p>Programming Notes:</p> <p>At least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).</p>
10	Blue Channel Mask
9	Green Channel Mask
8	Red Channel Mask

Typed Surface Read/Write Message Descriptor

Bit	Description
13:12	<p>Slot Group</p> <p>This field controls which 8 bits of Pixel/Sample Mask in the message header are ANDed with the execution mask to determine which slots are accessed. This field is ignored if the header is not present.</p> <p>Format = U2</p> <p>00: SIMD4x2</p> <p>01: Use low 8 slots</p> <p>10: Use high 8 slots</p> <p>11: Reserved</p>
11	<p>Alpha Channel Mask</p> <p>For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface.</p> <p>0: Alpha channel included</p> <p>1: Alpha channel not included</p> <p>Programming Notes:</p> <p>At least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).</p>
10	Blue Channel Mask
9	Green Channel Mask
8	Red Channel Mask

Untyped Surface ReadWrite Message Descriptor

Bits	Description
13:12	<p>SIMD Mode Format = U2 0: SIMD4x2 (valid for reads & writes) 1: SIMD16 2: SIMD8 3: Reserved</p>
11	<p>Alpha Channel Mask For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface. 0: Alpha channel included 1: Alpha channel not included Programming Notes: For the <i>Untyped Surface Write</i> message, each channel mask cannot be 0 unless all of the lower mask bits are also zero. This means that the only 4-bit channel mask values allowed are 0000b, 1000b, 1100b, and 1110b. Other messages allow any combination of channel masks. For the <i>Untyped Surface Read</i> message, at least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).</p>
10	Blue Channel Mask
9	Green Channel Mask
8	Red Channel Mask

Typed Atomic Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Slot Group This field controls which 8 bits of Pixel/Sample Mask in the message header are ANDed with the execution mask to determine which slots are accessed. Format = U1 0: Use low 8 slots 1: Use high 8 slots
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

Typed Atomic Operation SIMD4x2 Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

Untyped Atomic Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	SIMD Mode Format = U1 0: SIMD16 1: SIMD8
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: 0000: AOP_CMPWR8B 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

Untyped Atomic Operation SIMD4x2 Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: AOP_CMPWR8B 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

Atomic Counter Operation Message Descriptor

Bits	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	SIMD Mode Format: U1 0: Reserved 1: SIMD8 (low 8 slots)
11:8	Atomic Operation Type Specifies the atomic operation to perform: 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: Reserved 1111: AOP_PREDEC

For Append Counter Operations there is no address payload as the address is provided by the append counter field in the surface state. The write data payloads are the same as untyped atomic. The write backs are the same as untyped atomic.

When accessing a surface with the Append Counter Operation, if the Append Counter enable field of the surface state is not 1, the access is treated as out of bounds, with writes ignored and reads returning 0.

Atomic Counter Operation SIMD4x2 Message Descriptor

Bits	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: Reserved 1111: AOP_PREDEC

For Append Counter Operations there is no address payload as the address is provided by the append counter field in the surface state. The write data payloads are the same as untyped atomic 4x2. The write backs are the same as untyped atomic 4x2.

When accessing a surface with the Append Counter Operation, if the Append Counter enable field of the surface state is not 1, the access is treated as out of bounds, with writes ignored and reads returning 0.

Message Header

The message header for the untyped messages only needs to be delivered for pixel shader threads, where the execution mask may indicate pixels/samples that are enabled only due to derivative (LOD) calculations, but the corresponding slot on the surface must not be accessed. Typed messages (which go to the render cache data port) must include the header.

DWord	Bits	Description
M0.7	31:16	Ignored
	15:0	Pixel/Sample Mask. This field contains the 16-bit pixel/sample mask to be used for SIMD16 and SIMD8 messages. All 16 bits are used for SIMD16 messages. For typed SIMD8 messages, Slot Group selects which 8 bits of this field are used. For untyped SIMD8 messages, the low 8 bits of this field are used. If the header is not delivered, this field defaults to all ones. The field is ignored for SIMD4x2 messages.
M0.6	31:0	Ignored
M0.5	31:0	Immediate Buffer Base Address. Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), else this field is ignored. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:10]
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

Message Payload

The message payload consists of the following:

- For the read messages, only an address payload is delivered.
- For the write messages, an address payload is followed by the write data payload.
- For the atomic operation messages, an address payload is followed by the source payload.
- For SIMD16 and SIMD8 messages, the message length is used to determine how many address parameters are included in the message. The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled, and the number of message registers in the source payload is determined by the atomic operation operation. Thus, one or neither of these two values (depending on the message type), plus one for the header, can be subtracted from the message length to determine the number of message registers in the address payload, from which the number of address parameters can be determined.

SIMD16 Address Payload

The payload of a SIMD16 message provides address parameters to process 16 slots. The possible address parameters are U and V (since SIMD16 is only supported with untyped messages). The number of parameters required depends on the surface type being accessed. Each parameter takes two message registers. 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.

Programming Note	
Context:	SIMD16 Address Payload
For untyped messages of surface type SURFTYPE_BUFFER, either U only can be sent or U and V can be sent. If V is sent it is ignored.	

DWord	Bit	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2.7	31:0	Slot 15 U Address
M2.6	31:0	Slot 14 U Address
M2.5	31:0	Slot 13 U Address
M2.4	31:0	Slot 12 U Address
M2.3	31:0	Slot 11 U Address
M2.2	31:0	Slot 10 U Address
M2.1	31:0	Slot 9 U Address
M2.0	31:0	Slot 8 U Address
M3		Slots 7:0 V Address
M4		Slots 15:8 V Address

SIMD16 Source Payload (Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0 Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M5.6	31:0	Slot 6 Source0
M5.5	31:0	Slot 5 Source0
M5.4	31:0	Slot 4 Source0
M5.3	31:0	Slot 3 Source0
M5.2	31:0	Slot 2 Source0
M5.1	31:0	Slot 1 Source0
M5.0	31:0	Slot 0 Source0
M6.7	31:0	Slot 15 Source0
M6.6	31:0	Slot 14 Source0
M6.5	31:0	Slot 13 Source0
M6.4	31:0	Slot 12 Source0
M6.3	31:0	Slot 11 Source0
M6.2	31:0	Slot 10 Source0
M6.1	31:0	Slot 9 Source0
M6.0	31:0	Slot 8 Source0
M7		Slots 7:0 Source1
M8		Slots 15:8 Source1

SIMD16 Source Payload (AOP_CMPWR8B Only)

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0[31:0] Specifies Source0[31:0] for slot 7. Format = U32
M5.6	31:0	Slot 6 Source0[31:0]
M5.5	31:0	Slot 5 Source0[31:0]
M5.4	31:0	Slot 4 Source0[31:0]
M5.3	31:0	Slot 3 Source0[31:0]
M5.2	31:0	Slot 2 Source0[31:0]
M5.1	31:0	Slot 1 Source0[31:0]
M5.0	31:0	Slot 0 Source0[31:0]
M6.7	31:0	Slot 15 Source0[31:0]
M6.6	31:0	Slot 14 Source0[31:0]
M6.5	31:0	Slot 13 Source0[31:0]
M6.4	31:0	Slot 12 Source0[31:0]
M6.3	31:0	Slot 11 Source0[31:0]
M6.2	31:0	Slot 10 Source0[31:0]
M6.1	31:0	Slot 9 Source0[31:0]
M6.0	31:0	Slot 8 Source0[31:0]
M7		Slots 7:0 Source0[63:32]
M8		Slots 15:8 Source0[63:32]
M9		Slots 7:0 Source1[31:0]
M10		Slots 15:8 Source1[31:0]
M11		Slots 7:0 Source1[63:32]
M12		Slots 15:8 Source1[63:32]

SIMD16 Write Data Payload (Write Message Only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included or if some of the channel mask bits are asserted. Any parameter or write channel not included in the payload is skipped, with message phases below it being renumbered to take up the vacated space.

DWord	Bit	Description
M5.7	31:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. Format = 32 bits raw data.
M5.6	31:0	Slot 6 Red
M5.5	31:0	Slot 5 Red
M5.4	31:0	Slot 4 Red
M5.3	31:0	Slot 3 Red
M5.2	31:0	Slot 2 Red
M5.1	31:0	Slot 1 Red
M5.0	31:0	Slot 0 Red
M6.7	31:0	Slot 15 Red
M6.6	31:0	Slot 14 Red
M6.5	31:0	Slot 13 Red
M6.4	31:0	Slot 12 Red
M6.3	31:0	Slot 11 Red
M6.2	31:0	Slot 10 Red
M6.1	31:0	Slot 9 Red
M6.0	31:0	Slot 8 Red
M7		Slots 7:0 Green
M8		Slots 15:8 Green
M9		Slots 7:0 Blue
M10		Slots 15:8 Blue
M11		Slots 7:0 Alpha
M12		Slots 15:8 Alpha

SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameters are U, V, R, and LOD. The number of parameters required depends on the surface type being accessed. Each parameter takes one message 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.

Programming Note	
Context:	SIMD8 Address Payload.htm
<ul style="list-style-type: none"> For untyped messages of surface type SURFTYPE_BUFFER, either U only can be sent or U and V can be sent. If V is sent it is ignored For untyped messages of surface type SURFTYPE_STRBUF, both U and V must be sent. 	

DWord	Bits	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2		Slots 7:0 V Address
M3		Slots 7:0 R Address Programming Notes: This register can only be delivered for the <i>Typed</i> message types.
M4		Slots 7:0 LOD Programming Notes: This register can only be delivered for the <i>Typed</i> message types.

SIMD8 Source Payload (Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0 Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M5.6	31:0	Slot 6 Source0
M5.5	31:0	Slot 5 Source0
M5.4	31:0	Slot 4 Source0
M5.3	31:0	Slot 3 Source0
M5.2	31:0	Slot 2 Source0
M5.1	31:0	Slot 1 Source0
M5.0	31:0	Slot 0 Source0
M6		Slots 7:0 Source1

SIMD8 Write Data Payload (Write Message Only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included or if some of the channel mask bits are asserted. Any parameter or write channel not included in the payload is skipped, with message phases below it being renumbered to take up the vacated space.

DWord	Bit	Description
M5.7	31:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For <i>Untyped</i> messages: Format = 32 bits raw data. For <i>Typed</i> messages: Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
M5.6	31:0	Slot 6 Red
M5.5	31:0	Slot 5 Red
M5.4	31:0	Slot 4 Red
M5.3	31:0	Slot 3 Red
M5.2	31:0	Slot 2 Red
M5.1	31:0	Slot 1 Red
M5.0	31:0	Slot 0 Red
M6		Slots 7:0 Green
M7		Slots 7:0 Blue
M8		Slots 7:0 Alpha

SIMD8 Write Data Payload (Tile W Write Message Only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included.

DWord	Bit	Description
M5.7	31:8	Ignored
	7:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For <i>Typed</i> messages: Format = U8
M5.6	31:8	Ignored
	7:0	Slot 6 Red
M5.5	31:8	Ignored
	7:0	Slot 5 Red
M5.4	31:8	Ignored
	7:0	Slot 4 Red
M5.3	31:8	Ignored
	7:0	Slot 3 Red
M5.2	31:8	Ignored
	7:0	Slot 2 Red
M5.1	31:8	Ignored
	7:0	Slot 1 Red
M5.0	31:8	Ignored
	7:0	Slot 0 Red

SIMD4x2 Address Payload

The payload of a SIMD4x2 message provides address parameters to process 2 slots.

DWord	Bits	Description	Project
M1.7	31:0	Slot 1 LOD Programming Note: This register can only be delivered for the <i>Typed</i> message types.	
M1.6	31:0	Slot 1 R Address Programming Note: This register can only be delivered for the <i>Typed</i> message types.	
M1.5	31:0	Slot 1 V Address Format = U32	
M1.4	31:0	Slot 1 U Address Format = U32	
M1.3	31:0	Slot 0 LOD	
M1.2	31:0	Slot 0 R Address	
M1.1	31:0	Slot 0 V Address	
M1.0	31:0	Slot 0 U Address	

SIMD4x2 Source Payload (Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M2.7	31:0	Ignored
M2.6	31:0	Ignored
M2.5	31:0	Slot 1 Source1 Specifies Source1 for slot 1. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M2.4	31:0	Slot 1 Source0
M2.3	31:0	Ignored
M2.2	31:0	Ignored
M2.1	31:0	Slot 0 Source1
M2.0	31:0	Slot 0 Source0

SIMD4x2 Source Payload (AOP_CMPWR8B Only)

DWord	Bit	Description
M2.7	31:0	Slot 1 Source1 [63:32]
M2.6	31:0	Slot 1 Source1 [31:0]
M2.5	31:0	Slot 1 Source0 [63:32]
M2.4	31:0	Slot 1 Source0 [31:0]
M2.3	31:0	Slot 0 Source1 [63:32]
M2.2	31:0	Slot 0 Source1 [31:0]
M2.1	31:0	Slot 0 Source0 [63:32]
M2.0	31:0	Slot 0 Source0 [31:0]

SIMD4x2 Write Data Payload (Write Message Only)

The write data payload follows the address payload for write messages.

DWord	Bit	Description
M2.7	31:0	Slot 1 Alpha Specifies the value of the red channel to be written for slot 7. For <i>Untyped</i> messages: Format = 32 bits raw data. For <i>Typed</i> messages: Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
M2.6	31:0	Slot 1 Blue
M2.5	31:0	Slot 1 Green
M2.4	31:0	Slot 1 Red
M2.3	31:0	Slot 0 Alpha
M2.2	31:0	Slot 0 Blue
M2.1	31:0	Slot 0 Green
M2.0	31:0	Slot 0 Red

Writeback Message

SIMD8 DWORD Read

DWord	Bit	Description
W0.7	31:0	DWord[Offset7]
W0.6	31:0	DWord[Offset6]
W0.5	31:0	DWord[Offset5]
W0.4	31:0	DWord[Offset4]
W0.3	31:0	DWord[Offset3]
W0.2	31:0	DWord[Offset2]
W0.1	31:0	DWord[Offset1]
W0.0	31:0	DWord[Offset0]

SIMD8 QWORD Read

DWord	Bit	Description
W0.7 W0.6	63:0	QWord[Offset3]
W0.5 W0.4	63:0	QWord[Offset2]
W0.3 W0.2	63:0	QWord[Offset1]
W0.1 W0.0	63:0	QWord[Offset0]
W1.7 W1.6	63:0	QWord[Offset7]
W1.5 W1.4	63:0	QWord[Offset6]
W1.3 W1.2	63:0	QWord[Offset5]
W1.1 W1.0	63:0	QWord[Offset4]

SIMD16 Read

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted 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 regid+1, and alpha to regid+2 and regid+3. The slots written within each destination register is determined by the execution mask on the "send" instruction.

DWord	Bit	Description
W0.7	31:0	Slot 7 Red: Specifies the value of the red channel for slot 7. Format = 32 bits raw data.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1.7	31:0	Slot 15 Red
W1.6	31:0	Slot 14 Red
W1.5	31:0	Slot 13 Red
W1.4	31:0	Slot 12 Red
W1.3	31:0	Slot 11 Red
W1.2	31:0	Slot 10 Red
W1.1	31:0	Slot 9 Red
W1.0	31:0	Slot 8 Red
W2		Slots 7:0 Green
W3		Slots 15:8 Green
W4		Slots 7:0 Blue
W5		Slots 15:8 Blue
W6		Slots 7:0 Alpha
W7		Slots 15:8 Alpha

SIMD8 Read

A SIMD8 writeback message consists of up to 4 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted 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 slots written within each destination register is determined by the execution mask on the "send" instruction.

DWord	Bit	Description
W0.7	31:0	Slot 7 Red: Specifies the value of the red channel for slot 7. For <i>Untyped</i> messages: Format = 32 bits raw data. For <i>Typed</i> messages: Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1		Slots 7:0 Green
W2		Slots 7:0 Blue
W3		Slots 7:0 Alpha

SIMD8 Read (Tile W)

The slots written within each destination register is determined by the execution mask on the “send” instruction.

DWord	Bit	Description
M5.7	31:8	Reserved (0)
	7:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For <i>Typed</i> messages: Format = U8
M5.6	31:8	Reserved (0)
	7:0	Slot 6 Red
M5.5	31:8	Reserved (0)
	7:0	Slot 5 Red
M5.4	31:8	Reserved (0)
	7:0	Slot 4 Red
M5.3	31:8	Reserved (0)
	7:0	Slot 3 Red
M5.2	31:8	Reserved (0)
	7:0	Slot 2 Red
M5.1	31:8	Reserved (0)
	7:0	Slot 1 Red
M5.0	31:8	Reserved (0)
	7:0	Slot 0 Red

SIMD4x2 Read

A SIMD4x2 writeback message always consists of a single message register containing all four color channels of each of the two slots. The 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 slot is asserted, that slot is considered to be active. The active channels in the channel mask will be written in the destination register for that slot. If the slot is inactive (all four execution mask bits deasserted), none of the channels for that slot will be written in the destination register.

DWord	Bit	Description
W0.7	31:0	Slot 1 Alpha: Specifies the value of the pixel’s alpha channel. Format = 32 bits raw data.
W0.6	31:0	Slot 1 Blue
W0.5	31:0	Slot 1 Green
W0.4	31:0	Slot 1 Red
W0.3	31:0	Slot 0 Alpha
W0.2	31:0	Slot 0 Blue
W0.1	31:0	Slot 0 Green
W0.0	31:0	Slot 0 Red

SIMD16 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the "send" instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	Slot 7 Return Data: Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data
W0.5	31:0	Slot 5 Return Data
W0.4	31:0	Slot 4 Return Data
W0.3	31:0	Slot 3 Return Data
W0.2	31:0	Slot 2 Return Data
W0.1	31:0	Slot 1 Return Data
W0.0	31:0	Slot 0 Return Data
W1.7	31:0	Slot 15 Return Data
W1.6	31:0	Slot 14 Return Data
W1.5	31:0	Slot 13 Return Data
W1.4	31:0	Slot 12 Return Data
W1.3	31:0	Slot 11 Return Data
W1.2	31:0	Slot 10 Return Data
W1.1	31:0	Slot 9 Return Data
W1.0	31:0	Slot 8 Return Data

SIMD16 Atomic Operation (AOP_CMPWR8B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	Slot 7 Return Data[31:0]: Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data[31:0]
W0.5	31:0	Slot 5 Return Data[31:0]
W0.4	31:0	Slot 4 Return Data[31:0]
W0.3	31:0	Slot 3 Return Data[31:0]
W0.2	31:0	Slot 2 Return Data[31:0]
W0.1	31:0	Slot 1 Return Data[31:0]
W0.0	31:0	Slot 0 Return Data[31:0]
W1.7	31:0	Slot 15 Return Data[31:0]
W1.6	31:0	Slot 14 Return Data[31:0]
W1.5	31:0	Slot 13 Return Data[31:0]
W1.4	31:0	Slot 12 Return Data[31:0]
W1.3	31:0	Slot 11 Return Data[31:0]
W1.2	31:0	Slot 10 Return Data[31:0]
W1.1	31:0	Slot 9 Return Data[31:0]
W1.0	31:0	Slot 8 Return Data[31:0]
W2		Slot 7:0 Return Data[63:32]
W3		Slot 15:8 Return Data[63:32]

SIMD8 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	Slot 7 Return Data: Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data
W0.5	31:0	Slot 5 Return Data
W0.4	31:0	Slot 4 Return Data
W0.3	31:0	Slot 3 Return Data
W0.2	31:0	Slot 2 Return Data
W0.1	31:0	Slot 1 Return Data
W0.0	31:0	Slot 0 Return Data

SIMD8 Atomic Operation (AOP_CMPWR8B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	Slot 7 Return Data[31:0]: Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data[31:0]
W0.5	31:0	Slot 5 Return Data[31:0]
W0.4	31:0	Slot 4 Return Data[31:0]
W0.3	31:0	Slot 3 Return Data[31:0]
W0.2	31:0	Slot 2 Return Data[31:0]
W0.1	31:0	Slot 1 Return Data[31:0]
W0.0	31:0	Slot 0 Return Data[31:0]
W1.7	31:0	Slot 7 Return Data[63:32]
W1.6	31:0	Slot 6 Return Data[63:32]
W1.5	31:0	Slot 5 Return Data[63:32]
W1.4	31:0	Slot 4 Return Data[63:32]
W1.3	31:0	Slot 3 Return Data[63:32]
W1.2	31:0	Slot 2 Return Data[63:32]
W1.1	31:0	Slot 1 Return Data[63:32]
W1.0	31:0	Slot 0 Return Data[63:32]

SIMD4x2 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

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	Slot 1 Return Data: Specifies the value of the return data for slot 1. Format = U32
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	Slot 0 Return Data

SIMD4x2 Atomic Operation (AOP_CMPWR8B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

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	Slot 1 Return Data: [63:32]
W0.4	31:0	Slot 1 Return Data: [31:0]
W0.3	31:0	reserved – not written to GRF
W0.2	31:0	reserved – not written to GRF
W0.1	31:0	Slot 0 Return Data: [63:32]
W0.0	31:0	Slot 0 Return Data[31:0]

A64 Scattered Read or Write

This message takes a set of offsets, and reads or writes SIMD8 scattered Qwords/Dwords/Bytes starting at each offset. The Qword form of this message requires aligned Qword and the Dword form requires aligned Dwords. The byte form of this message takes a set of offsets, and reads or writes simd8 scattered and possibly misaligned bytes, words, dwords or qwords starting at each offset.

Programming Note	
Context:	A64 Scattered Read or Write
Writes to overlapping addresses will have undefined write ordering. Hardware does check for and optimize for cases where offsets are equal or contiguous, however for optimal performance in some these cases a different message may provide higher performance.	

Programming Note	
Context:	A64 Scattered Read or Write
Only stateless message can be used. The message header is forbidden these A64 messages, and the offsets are from a base of absolute 0	

Applications:

SIMD8 reads where the indices are different (read one channel per message)

SIMD8 writes where the indices are different (write one channel per message)

general purpose QWord/Dword/Bytes scatter/gathering, used by media

Execution Mask. The low 8 bits execution mask are used to determine which Qwords/Dwords/Bytes are read into the destination GRF register (for read), or which Qwords/Dwords/Bytes are written to the surface (for write).

Canonical address check, if post address computation, bits [63:48] don't match bit [47] then a general protect fault will occurs. If current mode is an non-faulting mode, this error will be treated as out of bounds.

Message Descriptor

Bits	Description
13	Reserved: MBZ
12	Ignored
11:10	Data Size. Specifies the data size for each slot: 0: 1 block 1: 2 blocks 2: 4 blocks 3: 8 blocks (not valid for QWords)
9:8	Block Size. Specifies the size of blocks to be read or written: 10: QWords 01: DWords 00: Bytes All other encodings are reserved.

Message Address Payload

DWord	Bit	Description
M1.7- M1.6	63:0	Offset 3. Specifies the Byte offset of QWord 3 Specifies the Byte offset of DWord 3 Specifies the Byte offset of Byte3 Format = U64 Qword Range = [0,00007FFFFFFFFF8h] or [FFFF800000000000,FFFFFFFFFFFFFFFF8h] Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh] Byte Range = [0,00007FFFFFFFFFh] or [FFFF800000000000,FFFFFFFFFFFFFFFFh]
M1.- M1.4	63:0	Offset 2
M1.3 - M1.2	63:0	Offset 1
M1.1 - M1.0	63:0	Offset 0
M2.7 - M2.6	63:0	Offset 7
M2.5 - M2.4	63:0	Offset 6
M2.3 - M2.2	63:0	Offset 5
M2.1 - M2.0	63:0	Offset 4

Additional Message Payload (QWord Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offset_n** referred to below is the **Offset *n*** in the address payload converted to units of QWords. The **QWord** array index is also in units of QWords.

DWord	Bit	Description
M3.7 - M3.6	63:0	QWord[Offset3][0]
M3.5 - M3.4	63:0	QWord[Offset2][0]
M3.3 - M3.2	63:0	QWord[Offset1][0]
M3.1 - M3.0	63:0	QWord[Offset0][0]
M4.7 - M4.6	63:0	QWord[Offset7][0]
M4.5 - M4.4	63:0	QWord[Offset6][0]
M4.3 - M4.2	63:0	QWord[Offset5][0]
M4.1 - M4.0	63:0	QWord[Offset4][0]
...
M9.7 - M9.6	63:0	QWord[Offset3][3]
M9.5 - M9.4	63:0	QWord[Offset2][3]
M9.3 - M9.2	63:0	QWord[Offset1][3]
M9.1 - M9.0	63:0	QWord[Offset0][3]
M10.7 - M10.6	63:0	QWord[Offset7][3]
M10.5 - M10.4	63:0	QWord[Offset6][3]
M10.3 - M10.2	63:0	QWord[Offset5][3]
M10.1 - M10.0	63:0	QWord[Offset4][3]

Additional Message Payload (DWord Write)

For the write operation, either one to four additional registers (depending on the block size) of payload contain the data to be written.

The **Offsetn** referred to below is the **Offset n** in the address payload converted to units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M3.7	31:0	DWord[Offset7][0]
M3.6	31:0	DWord[Offset6][0]
M3.5	31:0	DWord[Offset5][0]
M3.4	31:0	DWord[Offset4][0]
M3.3	31:0	DWord[Offset3][0]
M3.2	31:0	DWord[Offset2][0]
M3.1	31:0	DWord[Offset1][0]
M3.0	31:0	DWord[Offset0][0]
M4.7	31:0	DWord[Offset7][1]
M4.6	31:0	DWord[Offset6][1]
M4.5	31:0	DWord[Offset5][1]
M4.4	31:0	DWord[Offset4][1]
M4.3	31:0	DWord[Offset3][1]
M4.2	31:0	DWord[Offset2][1]
M4.1	31:0	DWord[Offset1][1]
M4.0	31:0	DWord[Offset0][1]
...
M9.7	31:0	DWord[Offset7][6]
M9.6	31:0	DWord[Offset6][6]
M9.5	31:0	DWord[Offset5][6]
M9.4	31:0	DWord[Offset4][6]
M9.3	31:0	DWord[Offset3][6]
M9.2	31:0	DWord[Offset2][6]
M9.1	31:0	DWord[Offset1][6]
M9.0	31:0	DWord[Offset0][6]
M10.7	31:0	DWord[Offset7][7]
M10.6	31:0	DWord[Offset6][7]
M10.5	31:0	DWord[Offset5][7]
M10.4	31:0	DWord[Offset4][7]
M10.3	31:0	DWord[Offset3][7]
M10.2	31:0	DWord[Offset2][7]

DWord	Bit	Description
M10.1	31:0	DWord[Offset1][7]
M10.0	31:0	DWord[Offset0][7]

Additional Message Payload (8 Byte Write)

For the write operation, two additional registers of payload contain the data to be written.

The **Offsetn** referred to below is **Offset n** in the address payload and is in units of bytes.

DWord	Bits	Description
M3.7	31:0	Byte[Offset3][7:4]
M3.6	31:0	Byte[Offset3][3:0]
M3.5	31:0	Byte[Offset2][7:4]
M3.4	31:0	Byte[Offset2][3:0]
M3.3	31:0	Byte[Offset1][7:4]
M3.2	31:0	Byte[Offset1][3:0]
M3.1	31:0	Byte[Offset0][7:4]
M3.0	31:0	Byte[Offset0][3:0]
M4.7	31:0	Byte[Offset7][7:4]
M4.6	31:0	Byte[Offset7][3:0]
M4.5	31:0	Byte[Offset6][7:4]
M4.4	31:0	Byte[Offset6][3:0]
M4.3	31:0	Byte[Offset5][7:4]
M4.2	31:0	Byte[Offset5][3:0]
M4.1	31:0	Byte[Offset4][7:4]
M4.0	31:0	Byte[Offset4][3:0]

Additional Message Payload (1 or 2 or 4 Byte Write)

For the write operation, two additional registers of payload contain the data to be written.

The **Offset_n** referred to below is **Offset n** in the address payload and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field. The upper bits are ignored for 1 byte and 2 byte **Data Size**.

DWord	Bits	Description
M3.7	31:0	Byte[Offset7][3:0]
M3.6	31:0	Byte[Offset6][3:0]
M3.5	31:0	Byte[Offset5][3:0]
M3.4	31:0	Byte[Offset4][3:0]
M3.3	31:0	Byte[Offset3][3:0]
M3.2	31:0	Byte[Offset2][3:0]
M3.1	31:0	Byte[Offset1][3:0]
M3.0	31:0	Byte[Offset0][3:0]

Writeback Message (QWord Read)

For the read operation, the writeback message consists of either two to eight registers depending on the **Data size**.

The **Offsetn** referred to below is the **Offset n** in the address payload converted to units of QWords. The **QWord** array index is also in units of QWords.

DWord	Bits	Description
W0.7 - W0.6	63:0	QWord[Offset3][0]
W0.5 - W0.4	63:0	QWord[Offset2][0]
W0.3 - W0.2	63:0	QWord[Offset1][0]
W0.1 - W0.0	63:0	QWord[Offset0][0]
W1.7 - W1.6	63:0	QWord[Offset7][0]
W1.5 - W1.4	63:0	QWord[Offset6][0]
W1.3 - W1.2	63:0	QWord[Offset5][0]
W1.1 - W1.0	63:0	QWord[Offset4][0]
...
W6.7 - W6.6	63:0	QWord[Offset3][3]
W6.5 - W6.4	63:0	QWord[Offset2][3]
W6.3 - W6.2	63:0	QWord[Offset1][3]
W6.1 - W6.0	63:0	QWord[Offset0][3]
W7.7 - W7.6	63:0	QWord[Offset7][3]
W7.5 - W7.4	63:0	QWord[Offset6][3]
W7.3 - W7.2	63:0	QWord[Offset5][3]
W7.1 - W7.0	63:0	QWord[Offset4][3]

Writeback Message (DWord Read)

For the read operation, the writeback message consists of either two to eight registers depending on the block size.

The **Offset_n** referred to below is the **Offset *n*** in the address payload converted to units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bits	Description
W0.7	31:0	DWord[Offset7][0]
W0.6	31:0	DWord[Offset6][0]
W0.5	31:0	DWord[Offset5][0]
W0.4	31:0	DWord[Offset4][0]
W0.3	31:0	DWord[Offset3][0]
W0.2	31:0	DWord[Offset2][0]
W0.1	31:0	DWord[Offset1][0]
W0.0	31:0	DWord[Offset0][0]
W1.7	31:0	DWord[Offset7][1]
W1.6	31:0	DWord[Offset6][1]
W1.5	31:0	DWord[Offset5][1]
W1.4	31:0	DWord[Offset4][1]
W1.3	31:0	DWord[Offset3][1]
W1.2	31:0	DWord[Offset2][1]
W1.1	31:0	DWord[Offset1][1]
W1.0	31:0	DWord[Offset0][1]
...
W6.7	31:0	DWord[Offset7][6]
W6.6	31:0	DWord[Offset6][6]
W6.5	31:0	DWord[Offset5][6]
W6.4	31:0	DWord[Offset4][6]
W6.3	31:0	DWord[Offset3][6]
W6.2	31:0	DWord[Offset2][6]
W6.1	31:0	DWord[Offset1][6]
W6.0	31:0	DWord[Offset0][6]
W7.7	31:0	DWord[Offset7][7]
W7.6	31:0	DWord[Offset6][7]
W7.5	31:0	DWord[Offset5][7]
W7.4	31:0	DWord[Offset4][7]
W7.3	31:0	DWord[Offset3][7]
W7.2	31:0	DWord[Offset2][7]

DWord	Bits	Description
W7.1	31:0	DWord[Offset1][7]
W7.0	31:0	DWord[Offset0][7]

Writeback Message (8 Byte Read)

For the read operation, the writeback message consists of two register.

The **Offsetn** referred to below is **Offset n** in the address payload and is in units of bytes.

DWord	Bits	Description
W0.7	31:0	Byte[Offset3][7:4]
W0.6	31:0	Byte[Offset3][3:0]
W0.5	31:0	Byte[Offset2][7:4]
W0.4	31:0	Byte[Offset2][3:0]
W0.3	31:0	Byte[Offset1][7:4]
W0.2	31:0	Byte[Offset1][3:0]
W0.1	31:0	Byte[Offset0][7:4]
W0.0	31:0	Byte[Offset0][3:0]
W1.7	31:0	Byte[Offset7][7:4]
W1.6	31:0	Byte[Offset7][3:0]
W1.5	31:0	Byte[Offset6][7:4]
W1.4	31:0	Byte[Offset6][3:0]
W1.3	31:0	Byte[Offset5][7:4]
W1.2	31:0	Byte[Offset5][3:0]
W1.1	31:0	Byte[Offset4][7:4]
W1.0	31:0	Byte[Offset4][3:0]

Writeback Message (1 or 2 or 4 Byte Read)

For the read operation, the writeback message consists of one register.

The **Offset_n** referred to below is **Offset n** in the address payload and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field and only the requested bytes are written to the GRF.

DWord	Bits	Description
W0.7	31:0	Byte[Offset7][3:0]
W0.6	31:0	Byte[Offset6][3:0]
W0.5	31:0	Byte[Offset5][3:0]
W0.4	31:0	Byte[Offset4][3:0]
W0.3	31:0	Byte[Offset3][3:0]
W0.2	31:0	Byte[Offset2][3:0]
W0.1	31:0	Byte[Offset1][3:0]
W0.0	31:0	Byte[Offset0][3:0]

A64 Untyped Atomic Operation

A64 Untyped Atomic Operation allows direct read/write stateless accesses.

This message supports 2 data sizes: DWORD and QWORD.

These messages use the RAW format, perform no type conversion. It uses just the U address parameter, which specifies the byte offset, which must be a multiple of 8 for QWORD and a multiple of 4 for DWORD. The atomic operation messages only access the first QWord.

The atomic operation messages cause atomic read-modify-write operations on the “destination” location addressed. In the table below, the new value of the destination (new_dst) is computed as indicated based on the old value of the destination (old_dst) and up to two sources included in the message (src0 and src1). Optionally, a value can be returned by the message (ret).

The atomic operations guarantee that the read and the write are performed atomically, meaning 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.

The following atomic operations are available for QWORD data size, along with the specific operation performed for each and the return value:

Atomic Operation	new_dst	ret
AOP_AND	old_dst & src0	old_dst
AOP_OR	old_dst src0	old_dst
AOP_XOR	old_dst ^ src0	old_dst
AOP_MOV	src0	old_dst
AOP_INC	old_dst + 1	old_dst
AOP_DEC	old_dst – 1	old_dst
AOP_ADD	old_dst + src0	old_dst
AOP_SUB	old_dst – src0	old_dst
AOP_REVSUB	src0 – old_dst	old_dst
AOP_IMAX	imax(old_dst, src0)	old_dst
AOP_IMIN	imin(old_dst, src0)	old_dst
AOP_UMAX	umax(old_dst, src0)	old_dst
AOP_UMIN	umin(old_dst, src0)	old_dst
AOP_CMPWR	(src0 == old_dst) ? src1 : old_dst	old_dst
AOP_PREDEC	old_dst – 1	new_dst
AOP_CMPWR16B	(src016B == old_dst16B) ? src116B : old_dst16B	old_dst16B

Programming Note	
Context:	A64 Untyped Atomic Operation
<ul style="list-style-type: none"> src016B is 16 bytes, src116B is 16 Bytes, and old_dst16B is 16 bytes in length. AOP_CMPWR16B is not supported for SLM. 	

Programming Note	
Context:	A64 Untyped Atomic Operation
<ul style="list-style-type: none"> AOP_CMPWR16B addresses must be OWORD-aligned. 	

Programming Note	
Context:	A64 Untyped Atomic Operation
AOP_CMPWR16B is not supported.	

Programming Note	
Context:	A64 Untyped Atomic Operation
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.	

The following atomic operations are available for DWORD data size, along with the specific operation performed for each and the return value:

Atomic Operation	new_dst	ret
AOP_AND	old_dst & src0	old_dst
AOP_OR	old_dst src0	old_dst
AOP_XOR	old_dst ^ src0	old_dst
AOP_MOV	src0	old_dst
AOP_INC	old_dst + 1	old_dst
AOP_DEC	old_dst - 1	old_dst
AOP_ADD	old_dst + src0	old_dst
AOP_SUB	old_dst - src0	old_dst
AOP_REVSUB	src0 - old_dst	old_dst
AOP_IMAX	imax(old_dst, src0)	old_dst
AOP_IMIN	imin(old_dst, src0)	old_dst
AOP_UMAX	umax(old_dst, src0)	old_dst
AOP_UMIN	umin(old_dst, src0)	old_dst
AOP_CMPWR	(src0 == old_dst) ? src1 : old_dst	old_dst
AOP_PREDEC	old_dst - 1	new_dst
AOP_CMPWR8B	(src08B == old_dst8B) ? src18B : old_dst8B	old_dst8B

Programming Note	
Context:	A64 Untyped Atomic Operation
<ul style="list-style-type: none"> src08B is 8 bytes, src18B is 8 Bytes, and old_dst8B is 8 bytes in length. AOP_CMPWR8B is not supported for SLM. AOP_CMPWR8B addresses must be QWORD-aligned. 	

Programming Note	
Context:	A64 Untyped Atomic Operation
AOP_CMPWR8B is not supported for A64 SIMD4x2. Use a SIMD8 message for AOP_CMPRW8B.	

Programming Note	
Context:	A64 Untyped Atomic Operation
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.	

The following messages are supported on BDW only.

Programming Note	
Context:	A64 Untyped Atomic Operation
<ul style="list-style-type: none"> Only stateless messages can be used. The stateless model support is limited to untyped messages. Furthermore, they are treated as SURFTYPE_BUFFER and Surface Format of RAW. The message header is forbidden in these A64 messages, and the offsets are from a base of absolute 0. 	

Execution Mask: The low 8 bits of the execution mask determine which QWords/DWords/Bytes are read into the destination GRF register (for read), or which QWords/DWords/Bytes are written to the surface (for write).

Canonical address check: If post address computation, bits [63:48] don't match bit [47] then a general protect fault occurs. If the current mode is a non-faulting mode, this error is treated as out of bounds.

Programming Note	
Context:	A64 Untyped Atomic Operation
Writes to overlapping addresses have undefined write ordering.	

A64 Untyped Atomic Operation Message Descriptor

Bits	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Data Size This field controls the data size of the operation Format = U1 0: DWORD size 1: QWORD
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: 0000: AOP_CMPWR16B / AOP_CMPWR8B 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

Message Payload

The message payload consists of the following:

For the read messages, only an address payload is delivered.

For the write messages, an address payload is followed by the write data payload.

For the atomic operation messages, an address payload is followed by the source payload.

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

SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameter is U. Each parameter takes one message register. Each parameter always takes a consistent position in the input payload.

DWord	Bit	Description
M1.7 M1.6	63:0	Slot 3 U address. Specifies the Byte offset of QWord 3 Specifies the Byte offset of DWord 3 Format = U64 Qword Range = [0,00007FFFFFFFFF8h] or [FFFF800000000000,FFFFFFFFFFFFFFFF8h] Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh]
M1.5 M1.4	63:0	Slot 2 U Address
M1.3 M1.2	63:0	Slot 1 U Address
M1.1 M1.0	63:0	Slot 0 U Address
M2.7 M2.6	63:0	Slot 7 U Address
M2.5 M2.4	63:0	Slot 6 U Address
M2.3 M2.2	63:0	Slot 5 U Address
M2.1 M2.0	63:0	Slot 4 U Address

SIMD4x2 Address Payload

The payload of a SIMD4x2 message provides address parameters to process 2 slots. The possible address parameter is U. Each parameter always takes a consistent position in the input payload.

DWord	Bit	Description
M1.7 M1.6	63:0	ignored
M1.5 M1.4	63:0	Slot 1 U Address Specifies the U Address for slot 1. Format = U64 Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh]
M1.3 M1.2	63:0	ignored
M1.1 M1.0	63:0	Slot 0 U Address

SIMD8 Source Payload (QWORD Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M3.7 - M3.6	63:0	Slot 3 Source0 Specifies Source0 for slot 3. Format = S63 for AOP_IMAX and AOP_IMIN, U64 for all other operations
M3.5 - M3.4	63:0	Slot 2 Source0
M3.3 - M3.2	63:0	Slot 1 Source0
M3.1 - M3.0	63:0	Slot 0 Source0
M4.7 - M4.6	63:0	Slot 7 Source0
M4.5 - M4.4	63:0	Slot 6 Source0
M4.3 - M4.2	63:0	Slot 5 Source0
M4.1 - M4.0	63:0	Slot 4 Source0
M5		Slot3:0 Source1
M6		Slot7:4 Source1

SIMD8 Source Payload (DWORD Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M3.7	31:0	Slot 7 Source0 Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M3.6	31:0	Slot 6 Source0
M3.5	31:0	Slot 5 Source0
M3.4	31:0	Slot 4 Source0
M3.3	31:0	Slot 3 Source0
M3.2	31:0	Slot 2 Source0
M3.1	31:0	Slot 1 Source0
M3.0	31:0	Slot 0 Source0
M4		Slots 7:0 Source1

SIMD8 Source Payload (AOP_CMPWR16B Only)

DWord	Bit	Description
M6.7 - M6.4	31:0	Slot 1 Source0[128:0] Specifies Source0[127:0] for slot 1. Format = U128
M6.3 - M6.0	31:0	Slot 0 Source0[127:0]
M7		Slot 3:2 Source0
M8		Slot 5:4 Source0
M9		Slot 7:6 Source0
M10		Slot 1:0 Source1
M11		Slot 3:2 Source1
M12		Slot 5:4 Source1
M13		Slot 7:6 Source1

SIMD8 Source Payload (AOP_CMPWR8B Only)

DWord	Bit	Description
M6.7 M6.6		Slot 3 Source0[63:0] Specifies Source0[63:0] for slot 3. Format = U64
M6.5 M6.4		Slot 2 Source0[63:0]
M6.3 M6.2		Slot 1 Source0
M6.1 M6.0		Slot 0 Source0
M7		Slot 7:4 Source0
M8		Slot 3:0 Source1
M9		Slot 7:4 Source1

SIMD4x2 Source Payload (QWORD Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M3.7 M3.6	63:0	ignored
M3.5 M3.4	63:0	Slot 1 Source0 Specifies Source0 for slot 1. Format = S63 for AOP_IMAX and AOP_IMIN, U64 for all other operations
M3.3 M3.2	63:0	ignored
M3.1 M3.0	63:0	Slot 0 Source0
M4		Slot1:0 Source1

SIMD4x2 Source Payload (DWORD Atomic Operation Message Only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M3.7	31:0	ignored
M3.6	31:0	ignored
M3.5	31:0	Slot 1 Source1
M3.4	31:0	Slot 1 Source0 Specifies Source0 for slot 1. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M3.3	31:0	ignored
M3.2	31:0	ignored
M3.1	31:0	Slot 0 Source1
M3.0	31:0	Slot0 Source0

SIMD4x2 Source Payload (AOP_CMPWR16B Only)

DWord	Bit	Description
M6.7 - M6.4	31:0	Slot 1 Source0[128:0] Specifies Source0[127:0] for slot 1. Format = U128
M6.3 - M6.0	31:0	Slot 0 Source0[127:0]
M7		Slot 1:0 Source1

SIMD4x2 Source Payload (AOP_CMPWR8B Only)

DWord	Bit	Description
M6.7 M6.6		ignored
M6.5 M6.4		Slot 1 Source0[63:0] Specifies Source0[63:0] for slot 1. Format = U64
M6.3 M6.2		ignored
M6.1 M6.0		Slot 0 Source0
M7		Slot 1:0 Source1

Writeback Message

The following subsections describe the writeback messages for A64 Untyped Atomic operations.

SIMD8 Atomic Operation (QWord)

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7 W0.6	63:0	Slot 3 Return Data: Specifies the value of the return data for slot 3. Format = U64
W0.5 W0.4	63:0	Slot 2 Return Data
W0.3 w0.2	63:0	Slot 1 Return Data
W0.1 W0.0	63:0	Slot 0 Return Data
W1.7 W1.6	63:0	Slot 7 Return Data
W1.5 W1.4	63:0	Slot 6 Return Data
W1.3 W1.2	63:0	Slot 5 Return Data
W1.1 W1.0	63:0	Slot 4 Return Data

SIMD8 Atomic Operation (DWord)

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	Slot 7 Return Data: Specifies the value of the return data for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
W0.6	31:0	Slot 6 Return Data
W0.5	31:0	Slot 5 Return Data
W0.4	31:0	Slot 4 Return Data
W0.3	31:0	Slot 3 Return Data
W0.2	31:0	Slot 2 Return Data
W0.1	31:0	Slot 1 Return Data
W0.0	31:0	Slot 0 Return Data

SIMD8 Atomic Operation (AOP_CMPWR16B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR16B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7 - W0.4		Slot 1 Return Data[127:0]: Specifies the value of the return data for slot 1. Format = U128
W0.3 - W0.0		Slot 0 Return Data[127:0]
W1		Slot 3:2 Return Data
W2		Slot 5:4 Return Data
W3		Slot 7:6 Return Data

SIMD8 Atomic Operation (AOP_CMPWR8B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction

DWord	Bit	Description
W0.7 W0.6	63:0	Slot 3 Return Data: Specifies the value of the return data for slot 3. Format = U64
W0.5 W0.4	63:0	Slot 2 Return Data
W0.3 w0.2	63:0	Slot 1 Return Data
W0.1 W0.0	63:0	Slot 0 Return Data
W1.7 W1.6	63:0	Slot 7 Return Data
W1.5 W1.4	63:0	Slot 6 Return Data
W1.3 W1.2	63:0	Slot 5 Return Data
W1.1 W1.0	63:0	Slot 4 Return Data

SIMD4x2 Atomic Operation (QWord)

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7 W0.6	63:0	reserved
W0.5 W0.4	63:0	Slot 1 Return Data: Specifies the value of the return data for slot 1. Format = U64
W0.3 w0.2	63:0	reserved
W0.1 W0.0	63:0	Slot 0 Return Data

SIMD4x2 Atomic Operation (DWord)

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W3.7	31:0	reserved
W3.6	31:0	reserved
W3.5	31:0	reserved
W3.4	31:0	Slot 1 Return Data: Specifies the value of the return data for slot 1. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
W3.3	31:0	reserved
W3.2	31:0	reserved
W3.1	31:0	reserved
W3.0	31:0	Slot 0 Return Data

SIMD4x2 Atomic Operation (AOP_CMPWR16B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR16B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7 - W0.4		Slot 1 Return Data[127:0]: Specifies the value of the return data for slot 1. Format = U128
W0.3 - W0.0		Slot 0 Return Data[127:0]

SIMD4x2 Atomic Operation (AOP_CMPWR8B Only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction

DWord	Bit	Description
W0.7 W0.6	63:0	reserved
W0.5 W0.4	63:0	Slot 1 Return Data: Specifies the value of the return data for slot 1. Format = U64
W0.3 w0.2	63:0	reserved
W0.1 W0.0	63:0	Slot 0 Return Data

A64 Untyped Atomic Float Add Operation

A64 Untyped Atomic Float Add Operation allows direct read/write stateless accesses.

This message supports 2 data sizes: DWORD and QWORD.

These messages use the RAW format, perform no type conversion. It uses just the U address parameter, which specifies the byte offset, which must be a multiple of 8 for QWORD and a multiple of 4 for DWORD. The atomic operation messages will only access the first QWORD.

The atomic operation messages cause atomic read-modify-write operations on the “destination” location addressed. In the table below, the new value of the destination (new_dst) is computed as indicated based on the old value of the destination (old_dst) and up to two sources included in the message (src0 and src1). Optionally, a value can be returned by the message (ret).

Programming Note	
Context:	A64 Untyped Atomic Float Add Operation
<ul style="list-style-type: none"> Only stateless message can be used. The message header is forbidden these A64 messages, and the offsets are from a base of absolute 0. 	

Canonical address check: If post address computation, bits [63:48] don't match bit [47] then a General Protect fault occurs. If the current mode is a non-faulting mode, this error is treated as out of bounds.

Untyped Atomic Float Add Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11	Data Size This field controls the data size of the operation Format = U1 0: DWORD size 1: QWORD
10:8	Reserved

Message Payload

SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameter is U. The number of parameters required depends on the surface type being accessed. Each parameter takes one message register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bit	Description
M1.7 M1.6	63:0	Slot 3 U address. Specifies the Byte offset of QWord 3 Specifies the Byte offset of DWord 3 Format = U64 Qword Range = [0,00007FFFFFFFFF8h] or [FFFF800000000000,FFFFFFFFFFFFFFFF8h] Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh]
M1.5 M1.4	63:0	Slot 2 U Address
M1.3 M1.2	63:0	Slot 1 U Address
M1.1 M1.0	63:0	Slot 0 U Address
M2.7 M2.6	63:0	Slot 7 U Address
M2.5 M2.4	63:0	Slot 6 U Address
M2.3 M2.2	63:0	Slot 5 U Address
M2.1 M2.0	63:0	Slot 4 U Address

SIMD8 Source Payload (QWORD Atomic Operation Message Only)

For the write operation two additional registers (depending on the block size) of payload contain the data to be written.

The **QWord** array index is also in units of QWords.

DWord	Bit	Description
M3.7 M3.6	63:0	Slot 3 Src0
M3.5 M3.4	63:0	Slot 2 Src0
M3.3 M3.2	63:0	Slot 1 Src0
M3.1 M3.0	63:0	Slot 0 Src0
M4.7 M4.6	63:0	Slot 7 Src0
M4.5 M4.4	63:0	Slot 6 Src0
M4.3 M4.2	63:0	Slot 5 Src0
M4.1 M4.0	63:0	Slot 4 Src0

SIMD8 Source Payload (DWORD Atomic Operation Message Only)

For the write operation two additional registers (depending on the block size) of payload contain the data to be written.

The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M3.7	31:0	Slot 7 Src0
M3.6	31:0	Slot 6 Src0
M3.5	31:0	Slot 5 Src0
M3.4	31:0	Slot 4 Src0
M3.3	31:0	Slot 3 Src0
M3.2	31:0	Slot 2 Src0
M3.1	31:0	Slot 1 Src0
M3.0	31:0	Slot 0 Src0

Writeback Message

A64 Untyped Surface Read or Write Operation

The A64 untyped surface message, both of which use the RAW surface format, perform no type conversions. A raw buffer uses just the U address parameter, which specifies the byte offset, which must be a multiple of 4.

For both raw messages up to 4 DWords are accessed beginning at the byte address determined. These 4 DWords correspond to the red, green, blue, and alpha channels in that order with red mapping to the lowest order DWord.

Programming Note	
Context:	A64 Untyped Surface Read or Write Operation
<ul style="list-style-type: none"> Only stateless message can be used. Furthermore, they are treated as SURFTYPE_BUFFER and Surface Format of RAW. For untyped messages accessing SURFTYPE_BUFFER, the U address (byte offset) must be DWord-aligned (low 2 bits must be zero). The message header is forbidden in these A64 messages, and the offsets are from a base of absolute 0. 	

Execution Mask:

SIMD8: The low 8 bits of the execution mask are used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write).

SIMD4x2: Each group of 4 bits within the low 8 bits of the execution mask are ORed together to create two bits used to determine which slots are read into the destination GRF register.

Canonical address check: if post address computation, bits [63:48] don't match bit [47] then a general protect fault occurs. If the current mode is a non-faulting mode, this error is treated as out of bounds.

Untyped Surface Read or Write Message Descriptor

Bits	Description
13:12	SIMD Mode Format = U2 0: SIMD4x2 1: Reserved 2: SIMD8 3: Reserved
11	Alpha Channel Mask For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface. 0: Alpha channel included. 1: Alpha channel not included. Programming Notes: For the <i>Untyped Surface Write</i> message, each channel mask cannot be 0 unless all of the lower mask bits are also zero. This means that the only 4-bit channel mask values allowed are 0000b, 1000b, 1100b, and 1110b. Other messages allow any combination of channel masks. For the <i>Untyped Surface Read</i> message, at least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).
10	Blue Channel Mask
9	Green Channel Mask
8	Red Channel Mask

Message Payload

The following subsections describe the message payloads for A64 Untyped Surface Read or Write Operations.

Read or write messages use the SIMD8 Address Payload or the SIMD4x2 Address Payload.

Only write messages use the SIMD8 DWORD Write Data Payload or the SIMD4x2 DWORD Write Data Payload.

SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameter is U. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bits	Description
M1.7 M1.6	63:0	Slot 3 U Address Specifies the U Address for slot 3. Format = U64 Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh]
M1.5 M1.4	63:0	Slot 2 U Address
M1.3 M1.2	63:0	Slot 1 U Address
M1.1 M1.0	63:0	Slot 0 U Address
M2.7 M2.6	63:0	Slot 7 U Address
M2.5 M2.4	63:0	Slot 6 U Address
M2.3 M2.2	63:0	Slot 5 U Address
M2.1 M2.0	63:0	Slot 4 U Address

SIMD4x2 Address Payload

The payload of a SIMD4x2 message provides address parameters to process 2 slots. The possible address parameter is U. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bits	Description
M1.7 M1.6	63:0	Ignored
M1.5 M1.4	63:0	Slot 1 U Address Specifies the U Address for slot 1. Format = U64 Dword Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh]
M1.3 M1.2	63:0	Ignored
M1.1 M1.0	63:0	Slot 0 U Address

SIMD8 DWORD Write Data Payload (Write Message Only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included or if some of the channel mask bits are asserted. Any parameter or write channel not included in the payload is skipped, with message phases below it being renumbered to take up the vacated space.

DWord	Bits	Description
M3.7	31:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For <i>Untyped</i> messages: Format = 32 bits raw data.
M3.6	31:0	Slot 6 Red
M3.5	31:0	Slot 5 Red
M3.4	31:0	Slot 4 Red
M3.3	31:0	Slot 3 Red
M3.2	31:0	Slot 2 Red
M3.1	31:0	Slot 1 Red
M3.0	31:0	Slot 0 Red
M4		Slots 7:0 Green
M5		Slots 7:0 Blue
M6		Slots 7:0 Alpha

SIMD4x2 DWORD Write Data Payload (Write Message Only)

The write data payload follows the address payload for write messages.

DWord	Bits	Description
M2.7	31:0	Slot 1 Alpha Specifies the alpha channel value to write for slot 1. For <i>Untyped</i> messages: Format = 32 bits raw data.
M2.6	31:0	Slot 1 Blue
M2.5	31:0	Slot 1 Green
M2.4	31:0	Slot 1 Red
M2.3	31:0	Slot 0 Alpha
M2.2	31:0	Slot 0 Blue
M2.1	31:0	Slot 0 Green
M2.0	31:0	Slot 0 Red

Writeback Message

For A64 Untyped Surface Read or Write operations the writeback messages are SIMD8 DWORD Read and SIMD4x2 DWORD Read.

SIMD8 DWORD Read

A SIMD8 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted 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 regid+1, and alpha to regid+2 and regid+3. The slots written within each destination register are determined by the execution mask on the "send" instruction.

DWord	Bits	Description
W0.7	31:0	Slot 7 Red: Specifies the value of the red channel for slot 7. For <i>Untyped</i> messages: Format = 32 bits raw data.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1		Slots 7:0 Green
W2		Slots 7:0 Blue
W3		Slots 7:0 Alpha

SIMD4x2 DWORD Read

DWord	Bits	Description
W0.7	31:0	Slot 1 Alpha Specifies the alpha channel value to write for slot 1. For <i>Untyped</i> messages: Format = 32 bits raw data.
W0.6	31:0	Slot 1 Blue
W0.5	31:0	Slot 1 Green
W0.4	31:0	Slot 1 Red
W0.3	31:0	Slot 0 Alpha
W0.2	31:0	Slot 0 Blue
W0.1	31:0	Slot 0 Green
W0.0	31:0	Slot 0 Red

A64 Block Read or Write Operation

Programming Note	
Context:	A64 Block Read or Write Operation
<ul style="list-style-type: none"> • OWord & Dual Oword offsets must be aligned on OWord boundaries. • Unaligned OWord offsets must be aligned on DWord boundaries. • HWord offsets must be aligned on HWord boundaries. 	

Execution Mask (OWord read/write). The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth (W1, W3 or M2, M4). For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, is used as the single group of four with behavior following that in the preceding paragraph.

The above behavior enables a SIMD16 thread to use the 8-OWord form of this message to access two channels (red and green) of a single scratch register across 16 pixels. A second message would access the other two channels (blue and alpha). The execution mask is used to ensure that data associated with inactive pixels are not overwritten.

Execution Mask (Dual Oword read/write). The low 8 bits of the execution mask are used to enable the 8 channels in the GRF registers returned for read, or each of the write registers sent. For reads, any mask bit asserted within a group of four causes the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

Execution Mask (Unaligned OWord read/write). The execution mask is ignored by this message sub type.

Execution Mask (HWord read/write). The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth (W1, W3 or M2, M4). For reads either any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register or the mask bit control corresponding DWORD based on the **HWord Read/Write Channel Mode**. For writes, either any mask bit asserted within a group of four will cause the entire OWord to be written or each mask bit is considered for its corresponding DWord written to the destination surface **HWord Read/Write Channel Mode**.

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, is used as the single group of four with behavior following that in the preceding paragraph.

Canonical address check: if post address computation, bits [63:48] don't match bit [47] then a general protect fault occurs. If current mode is a non-faulting mode, this error is treated as out of bounds.

Message Descriptor

Bits	Description
13	Reserved: MBZ
12:11	Message sub-type: 00: OWord Block Read/Write 01: Unaligned OWord Block Read 10: OWord Dual Block Read/Write 11: HWord Block Read/Write
10:8	Block Size. Specifies the number of elements transferred. See table below.

	00	01	10	11
Block Size	Oword	UnAligned Oword	Oword Dual	Hword
000	1 OWord, read into or written from the low 128 bits of the destination register	1 OWord, read into or written from the low 128 bits of the destination register	reserved	reserved
001	1 OWord, read into or written from the high 128 bits of the destination register	1 OWord, read into or written from the high 128 bits of the destination register	1 Oword	1 HWord
010	2 OWords	2 OWords	reserved	2 HWord
011	4 OWords	4 OWords	4 OWords	4 HWord
100	8 OWords	8 OWords	reserved	8 HWord
101	reserved	reserved	reserved	reserved
110	reserved	reserved	reserved	reserved
111	reserved	reserved	reserved	reserved

Message Header

DWord	Bits	Description
M0.5	31	HWord Read/Write Channel Mode. This field is only used for HWord read/write messages. 0: OWord – Channel enables in effect at the time of 'send' are interpreted such that if one or more are enabled, the read or write operation occurs on all four DWords. 1: DWord – Channel enables in effect at the time of the 'send' are used as DWord enables, causing the read or write operation to occur only on the DWords where the corresponding channel enable is set.
	30:0	Ignored
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3 M0.2		Block Offset 1. Specifies the Byte offset of OWord Block 1 for OWord Dual reads. Format = U64 Dual OWord Range = [0,00007FFFFFFFFF0h] or [FFFF800000000000,FFFFFFFFFFFFFFFF0h]
M0.1 M0.0		Block Offset 0. Specifies the Byte offset of Block 0. Format = U64 Unaligned OWord Range = [0,00007FFFFFFFFFCh] or [FFFF800000000000,FFFFFFFFFFFFFFFFCh] Dual OWord Range = [0,00007FFFFFFFFF0h] or [FFFF800000000000,FFFFFFFFFFFFFFFF0h] OWord Range = [0,00007FFFFFFFFF0h] or [FFFF800000000000,FFFFFFFFFFFFFFFF0h] HWord Range = [0,00007FFFFFFFFFE0h] or [FFFF800000000000,FFFFFFFFFFFFFFFFE0h]

Message Payload (OWord Write)

For the write operation, the message payload consists of one, two, or four registers (not including the header) depending on the **Block Size** specified in the message. For the one-constant case, data is taken from either the high or low half of the payload register depending on the half selected in **Block Size**. In this case, the other half of the payload register is ignored.

DWord	Bits	Description
M1.7:4	127:0	OWord[Offset + 1]. If the block size is 1 OWord to be written from the high 128 bits of the destination, OWord[Offset] appears in this location.
M1.3:0	127:0	OWord[Offset]
M2.7:4	127:0	OWord[Offset+3]
M2.3:0	127:0	OWord[Offset+2]
M3.7:4	127:0	OWord[Offset+5]
M3.3:0	127:0	OWord[Offset+4]
M4.7:4	127:0	OWord[Offset+7]
M4.3:0	127:0	OWord[Offset+6]

Writeback Message (OWord Read)

For the read operation, the writeback message consists of one, two, three, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

DWord	Bits	Description
W0.7:4	127:0	OWord[Offset+1] . If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord[Offset] appears in this location.
W0.3:0	127:0	OWord[Offset]
W1.7:4	127:0	OWord[Offset+3]
W1.3:0	127:0	OWord[Offset+2]
W2.7:4	127:0	OWord[Offset+5]
W2.3:0	127:0	OWord[Offset+4]
W3.7:4	127:0	OWord[Offset+7]
W3.3:0	127:0	OWord[Offset+6]

Writeback Message (Unaligned OWord Read)

For the read operation, the writeback message consists of one, two, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

DWord	Bits	Description
W0.7:4	127:0	OWord1 = *(&OWord0 + 1) . If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord0 appears in this location.
W0.3:0	127:0	OWord0 = *Offset
W1.7:4	127:0	OWord3 = *(&OWord2 + 1)
W1.3:0	127:0	OWord2 = *(&OWord1 + 1)
W2.7:4	127:0	OWord5 = *(&OWord4 + 1)
W2.3:0	127:0	OWord4 = *(&OWord3 + 1)
W3.7:4	127:0	OWord7 = *(&OWord6 + 1)
W3.3:0	127:0	OWord6 = *(&OWord5 + 1)

Message Payload (Dual OWord Write)

For the write operation, the message payload consists of one or four registers (not including the header or the first part of the payload) depending on the **Block Size** specified in the message.

DWord	Bits	Description
M2.7:4	127:0	OWord[Offset1]
M2.3:0	127:0	OWord[Offset0]
M3.7:4	127:0	OWord[Offset1+1]
M3.3:0	127:0	OWord[Offset0+1]
M4.7:4	127:0	OWord[Offset1+2]
M4.3:0	127:0	OWord[Offset0+2]
M4.7:4	127:0	OWord[Offset1+3]
M4.3:0	127:0	OWord[Offset0+3]

Writeback Message (Dual Oword Read)

For the read operation, the writeback message consists of one or four registers depending on the **Block Size** specified in the message.

DWord	Bits	Description
W0.7:4	127:0	OWord[Offset1]
W0.3:0	127:0	OWord[Offset0]
W1.7:4	127:0	OWord[Offset1+1]
W1.3:0	127:0	OWord[Offset0+1]
W2.7:4	127:0	OWord[Offset1+2]
W2.3:0	127:0	OWord[Offset0+2]
W3.7:4	127:0	OWord[Offset1+3]
W3.3:0	127:0	OWord[Offset0+3]

Message Payload (HWord Write)

The table below illustrates the write payload for a message of block size = 4.

DWord	Bits	Description
M1.7:0	255:0	HWord[Offset]
M2.7:0	255:0	HWord[Offset+1]
M3.7:0	255:0	HWord[Offset+2]
M3.7:0	255:0	HWord[Offset+3]

Writeback Message (HWord Read)

The table below illustrates an example where 4 HWords are read through a scratch block read.

DWord	Bits	Description
W0.7:0	255:0	HWord[Offset]
W1.7:0	255:0	HWord[Offset+1]
W2.7:0	255:0	HWord[Offset+2]
W3.7:0	255:0	HWord[Offset+3]

Untyped Atomic Float Add Operation

Untyped Atomic Operation allows direct read/write accesses to surfaces. The categories of surface:

Raw buffer (untyped). These surfaces are of type SURFTYPE_BUFFER and have a surface format of RAW and a surface pitch of 1 byte. Supported via the data cache data port.

Structured buffer (untyped). These surfaces are of type SURFTYPE_STRBUF and have a surface format of RAW. Supported via the data cache data port.

This message supports 2 data sizes: Single Precision (DWORD) and Double Precision (QWORD).

The untyped surface categories, both of which use the RAW surface format, perform no type conversion. A raw buffer uses just the U address parameter, which specifies the byte offset into the surface, which must be a multiple of 8 for QWORD and a multiple of 4 for DWORD. For structured buffers, the U parameter is the index into an array of structures and the V parameter is the actual byte offset into the structure instance at index U, which must be DWord/QWord aligned as the case may be. The atomic operation messages only access the first QWord or DWord.

The atomic operation messages cause atomic read-modify-write operations on the destination location addressed. The new value of the destination (new_dest) is computed based on the old value of the destination (old_dest) by adding to it the source value (src0) included in the message. Optionally, the old value can be returned by the message (ret).

The atomic operations guarantee that the read and the write are performed atomically, meaning 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.

Execution Mask:

SIMD16: The 16 bits of the execution mask are ANDed with the 16 bits of the **Pixel/Sample Mask** from the message header and the resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the execution mask is used.

SIMD8: The low 8 bits of the execution mask are ANDed with 8 bits of the **Pixel/Sample Mask** from the message header. The resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the low 8 bits of the execution mask are used.

Out-of-Bounds Accesses: Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

Programming Note	
Context:	Untyped Atomic Float Add Operation
<ul style="list-style-type: none"> Writes to overlapping addresses will have undefined write ordering. The stateless model support is restricted to SURFTYPE_BUFFER and Surface Format of RAW. The bounds checking for the stateless message is 4GB overflow and < General State upper bound. 	

Untyped Atomic Float Add Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	SIMD Mode Format = U1 0: SIMD16 1: SIMD8
11	Data Size This field controls the data size of the operation Format = U1 0: DWORD size 1: QWORD
10:8	Reserved

Message Header

The message header for the untyped messages only needs to be delivered for pixel shader threads, where the execution mask may indicate pixels/samples that are enabled only due to derivative (LOD) calculations, but the corresponding slot on the surface must not be accessed.

DWord	Bit	Description
M0.7	31:16	Ignored
	15:0	Pixel/Sample Mask. This field contains the 16-bit pixel/sample mask to be used for SIMD16 and SIMD8 messages. All 16 bits are used for SIMD16 messages. For untyped SIMD8 messages, the low 8 bits of this field are used. If the header is not delivered, this field defaults to all ones. The field is ignored for SIMD4x2 messages.
M0.6	31:0	Ignored
M0.5	31:0	Immediate Buffer Base Address. Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:10]
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

Message Payload

SIMD16 Address Payload

The payload of a SIMD16 message provides address parameters to process 16 slots. The possible address parameters are U and V (since SIMD16 is only supported with untyped messages). The number of parameters required depends on the surface type being accessed. Each parameter takes two message registers. 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.

Programming Note	
Context:	SIMD16 Address Payload
For untyped messages of surface type SURFTYPE_BUFFER, either U only can be sent or U and V can be sent. If V is sent it is ignored.	

DWord	Bit	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2.7	31:0	Slot 15 U Address
M2.6	31:0	Slot 14 U Address
M2.5	31:0	Slot 13 U Address
M2.4	31:0	Slot 12 U Address
M2.3	31:0	Slot 11 U Address
M2.2	31:0	Slot 10 U Address
M2.1	31:0	Slot 9 U Address
M2.0	31:0	Slot 8 U Address
M3		Slots 7:0 V Address
M4		Slots 15:8 V Address

SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameters are U, V. The number of parameters required depends on the surface type being accessed. Each parameter takes one message 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.

Programming Note	
Context:	SIMD8 Address Payload
<ul style="list-style-type: none"> For untyped messages of surface type SURFTYPE_BUFFER, either U only can be sent or U and V can be sent. If V is sent it is ignored. For untyped messages of surface type SURFTYPE_STRBUF, both U and V must be sent. 	

DWord	Bit	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2		Slots 7:0 V Address

SIMD16/SIMD8 DWORD Source Payload

Either one or two additional registers (depending on the SIMD mode) of payload contain the sources to be used.

DWord	Bit	Description
M3.7	31:0	DWord[slot7]
M3.6	31:0	DWord[slot6]
M3.5	31:0	DWord[slot5]
M3.4	31:0	DWord[slot4]
M3.3	31:0	DWord[slot3]
M3.2	31:0	DWord[slot2]
M3.1	31:0	DWord[slot1]
M3.0	31:0	DWord[slot0]
M4.7	31:0	DWord[slot15]. This message register is included only for SIMD16
M4.6	31:0	DWord[slot14]
M4.5	31:0	DWord[slot13]
M4.4	31:0	DWord[slot12]
M4.3	31:0	DWord[slot11]
M4.2	31:0	DWord[slot10]
M4.1	31:0	DWord[slot9]
M4.0	31:0	DWord[slot8]

SIMD16/SIMD8 QWORD Source Payload

Either two or four additional registers (depending on the SIMD mode) of payload contain the sources to use.

DWord	Bits	Description
M3.7 M3.6	63:0	QWord[slot3]
M3.5 M3.4	63:0	QWord[slot2]
M3.3 M3.2	63:0	QWord[slot1]
M3.1 M3.0	63:0	QWord[slot0]
M4.7 M4.6	63:0	QWord[slot7]
M4.5 M4.4	63:0	QWord[slot6]
M4.3 M4.2	63:0	QWord[slot5]
M4.1 M4.0	63:0	QWord[slot4]
M5		Qword[slot11:slot8]. This register is only included for SIMD16.
M6		Qword[slot15:slot12]. This register is only included for SIMD16.

Writeback Message

SIMD16 SIMD8 DWORD Atomic Float Add

DWord	Bit	Description
W0.7	31:0	DWord[slot7]
W0.6	31:0	DWord[slot6]
W0.5	31:0	DWord[slot5]
W0.4	31:0	DWord[slot4]
W0.3	31:0	DWord[slot3]
W0.2	31:0	DWord[slot2]
W0.1	31:0	DWord[slot1]
W0.0	31:0	DWord[slot0]
W1.7	31:0	DWord[slot15]. This message register is included only for SIMD16.
W1.6	31:0	DWord[slot14]
W1.5	31:0	DWord[slot13]
W1.4	31:0	DWord[slot12]
W1.3	31:0	DWord[slot11]
W1.2	31:0	DWord[slot10]
W1.1	31:0	DWord[slot9]
W1.0	31:0	DWord[slot8]

SIMD16/SIMD8 QWORD Atomic Float Add

DWord	Bit	Description
W0.7/W0.6	63:0	QWord[slot3]
W0.5/W0.4	63:0	QWord[slot2]
W0.3/W0.2	63:0	QWord[slot1]
W0.1/W0.0	63:0	QWord[slot0]
W1.7/W1.6	63:0	QWord[slot7]
W1.5/W1.4	63:0	QWord[slot6]
W1.3/W1.2	63:0	QWord[slot5]
W1.1/W1.0	63:0	QWord[slot4]
W2		Qword[slot11:Offset8] . This register is only included for SIMD16.
W3		Qword[slot15:Offset12] . This register is only included for SIMD16.

Read Surface Info

This message is used to determine information about a surface.

Message Descriptor

Bits	Description
13:8	Reserved

Address Payload

Dword	Bits	Description
M1.7	31:0	Reserved
M1.6	31:0	Reserved
M1.5	31:0	Reserved
M1.4	31:0	Reserved
M1.3	31:0	LOD Address
M1.2	31:0	R Address
M1.1	31:0	V Address
M1.0	31:0	U Address

Writeback Message

W0.7	31:0	Instruction Base Address [63:32]	Instruction Base Address [63:32]	Instruction Base Address [63:32]	Instruction Base Address [63:32]	Instruction Base Address [63:32]	Instruction Base Address [63:32]	Instruction Base Address [63:32]
W0.6	31:0	Instruction Base Address [31:0]	Instruction Base Address [31:0]	Instruction Base Address [31:0]	Instruction Base Address [31:0]	Instruction Base Address [31:0]	Instruction Base Address [31:0]	Instruction Base Address [31:0]
W1.5	31:0	0	Surface Format	Surface Format	Surface Format	Surface Format	Surface Format	Surface Format
W1.4	31:0	SURFTYPE_NULL (7h)	SURFTYPE_STRBUF (5h)	SURFTYPE_BUFFER (4h)	SURFTYPE_CUBE (3h)	SURFTYPE_3D (2h)	SURFTYPE_2D (1h)	SURFTYPE_1D (0h)
W1.3	31:0	0	0	0	MIPCount	MIPCount	MIPCount	MIPCount
W1.2	31:0	0	0	0	Surface Array ? Depth+1 : 0	(Depth+1)>>LOD	Surface Array ? Depth+1 : 0	0
W1.1	31:0	0	(Height+1)>>LOD	0	(Height+1)>>LOD	(Height+1)>>LOD	(Height+1)>>LOD	Surface Array ? Depth+1 : 0
W1.0	31:0	0	(Width+1)>>LOD	(Width+1)>>LOD	(Width+1)>>LOD	(Width+1)>>LOD	(Width+1)>>LOD	(Width+1)>>LOD

The 64-bit Instruction Base Address is specified as a 48-bit state base address and is extended to 64 bits in HW. It is reflected here so SW can read it for conversion of 64-bit instruction pointers.

LOD Information

The LOD is in-bounds if $LOD < MIPCount$ and if $MinLOD + LOD < 15$. If LOD is not in-bounds then 0 is returned for the width, height, and depth values.

Transpose Read

This message enables a rectangular block of DWords to be read from the source surface and written into the GRF.

Programming Note	
Context:	Transpose Read
<p>Accesses are allowed to SURFTYPE_NULL, reads return 0 and writes are ignored.</p> <ul style="list-style-type: none"> • The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Because of this, the stateless surface model is not supported with this message. • The raw data from the surface is returned to the thread without any format conversion nor filtering operation. • The target cache must be the data cache. • The surface base address must be 32-byte aligned. • The surface width must be a multiple of DWORDs. • The only tiling modes supported are: Tile Y and Linear. • The only tiling modes supported are: Tile Y and Linear. • The block width must be 1, 2, 4, or 8 DWORDs. • The X Offset must be a multiple of the block width in bytes. • The surface format must be 32 bpp (DWORD). • Vertical stride, offset and boundary clamping modes from surface state are supported. • The block height must be 1, 2, 4, or 8 rows. • The Y Offset must be a multiple of the block height. • Pitch must be a multiple of 64 bytes when the surface is linear. 	

Applications:

Block reads/writes for media.

Execution Mask. The execution mask on the send instruction for this type of message is ignored. The data that is read is determined completely by the block parameters.

Out-of-Bounds Accesses. Reads outside of the surface results in the address being clamped to the nearest edge of the surface and the pixel in the position being returned.

Message Descriptor

Bits	Description
13:8	Reserved

Message Header

DWord	Bits	Description										
M0.5	31:8	Ignored										
	7:0	FFTID. 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.										
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)										
M0.3	31:0	Ignored										
M0.2	31:22	Ignored										
	21:20	Block Height. Height in rows of block being accessed.										
		<table><tr><th>Value</th><th>Block Height (Rows)</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>4</td></tr><tr><td>3</td><td>8</td></tr></table>	Value	Block Height (Rows)	0	1	1	2	2	4	3	8
		Value	Block Height (Rows)									
		0	1									
1		2										
2	4											
3	8											
Format = U2												
Range = [0,3] representing 1 to 8 rows												
19:2	Ignored											
1:0	Block Width. Width in DWORDs of block being accessed.											
		<table><tr><th>Value</th><th>Block Width (DWORDs)</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>4</td></tr><tr><td>3</td><td>8</td></tr></table>	Value	Block Width (DWORDs)	0	1	1	2	2	4	3	8
		Value	Block Width (DWORDs)									
		0	1									
		1	2									
2	4											
3	8											
Format = U2												
Range = [0,3] representing 1 to 8 DWORDs												
M0.1	31:0	Y offset. The Y offset (in rows) of the upper left corner of the block into the surface. Format = S31 Programming Note: This field must be a multiple of the block height.										
M0.0	31:0	X offset. The X offset (in bytes) of the upper left corner of the block into the surface. Format = S31 Programming Notes: This field must be a multiple of Block Width. Must be DWord aligned.										

Writeback Message

DWord	Bits	Description
W0.7	31:0	DWord[XOffset+0,YOffset+7]
W0.6	31:0	DWord[XOffset+0,YOffset+6]
W0.5	31:0	DWord[XOffset+0,YOffset+5]
W0.4	31:0	DWord[XOffset+0,YOffset+4]
W0.3	31:0	DWord[XOffset+0,YOffset+3]
W0.2	31:0	DWord[XOffset+0,YOffset+2]
W0.1	31:0	DWord[XOffset+0,YOffset+1]
W0.0	31:0	DWord[XOffset+0,YOffset+0]
W1.7	31:0	DWord[XOffset+1,YOffset+7]
W1.6	31:0	DWord[XOffset+1,YOffset+6]
W1.5	31:0	DWord[XOffset+1,YOffset+5]
W1.4	31:0	DWord[XOffset+1,YOffset+4]
W1.3	31:0	DWord[XOffset+1,YOffset+3]
W1.2	31:0	DWord[XOffset+1,YOffset+2]
W1.1	31:0	DWord[XOffset+1,YOffset+1]
W1.0	31:0	DWord[XOffset+1,YOffset+0]
...
W7.7	31:0	DWord[XOffset+7,YOffset+7]
W7.6	31:0	DWord[XOffset+7,YOffset+6]
W7.5	31:0	DWord[XOffset+7,YOffset+5]
W7.4	31:0	DWord[XOffset+7,YOffset+4]
W7.3	31:0	DWord[XOffset+7,YOffset+3]
W7.2	31:0	DWord[XOffset+7,YOffset+2]
W7.1	31:0	DWord[XOffset+7,YOffset+1]
W7.0	31:0	DWord[XOffset+7,YOffset+0]

Scratch Block Read or Write

This message performs a read or write operation of between 1 and 4 SIMD8 registers to an HWord aligned offset to scratch memory. The HWord offset into the scratch memory is provided in the message descriptor, allowing a single instruction read/write block operation in a single source instruction. 12 bits are provided for the HWord offset, allowing addressing of 4K Hword locations (128KB).

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. For

operations using SIMD32 processing, two messages should be used, with one of them indicating 'H2' to select the upper 16b of the execution mask.

This message type can only be used with stateless model memory access. Thus binding table entry 0xFF is hard-coded into the execution of this message.

Applications: Scratch space reads/writes for register spill/fill operations.

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth registers (W1, W3 or M2, M4).

For DWord mode, the execution mask delivered with the message dictates DWord-based control of read or write operations. For OWord 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.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and do not modify memory contents.

Programming Note	
Context:	Scratch Block Read or Write
Out-of-Bounds check is disabled.	

Message Descriptor

Bits	Description
17	Operation Type: 0 = Read, 1 = Write
16	Channel Mode: 0: OWord – Channel enables in effect at the time of 'send' are interpreted such that, if one or more are enabled, the read or write operation occurs on all four DWords. 1: DWord – Channel enables in effect at the time of the 'send' are used as DWord enables, causing the read or write operation to occur only on the DWords whose corresponding channel enable is set.
15	Reserved: MBZ.
14	Reserved: MBZ
13:12	Block Size. Indicates the number of SIMD8 registers to be read or written: 00: 1 register 01: 2 registers 10: 4 registers 11: 8 registers
11:0	Offset. A 12-bit HWord offset into the memory Immediate Memory buffer as specified by binding table 0xFF.

Message Header

DWord	Bits	Description
M0.7	31:16	Ignored
	15:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	Immediate Buffer Base Address. Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model); otherwise this field is ignored. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:10]
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

Message Payload (Write)

The table below illustrates the write payload for a message of block size = 4.

DWord	Bits	Description
M1.7:0	255:0	HWord[Offset]
M2.7:0	255:0	HWord[Offset+1]
M3.7:0	255:0	HWord[Offset+2]
M3.7:0	255:0	HWord[Offset+3]

Message Payload (Read)

Read only requires a message header and has no message address payload.

Writeback Message (Read)

The table below illustrates an example where 4 HWords are read through a scratch block read.

DWord	Bits	Description
W0.7:0	255:0	HWord[Offset]
W1.7:0	255:0	HWord[Offset+1]
W2.7:0	255:0	HWord[Offset+2]
W3.7:0	255:0	HWord[Offset+3]

Memory Fence

A memory fence message issued by a thread causes further messages issued by the thread to be blocked until all previous messages issued by the thread to that data port (data cache or render cache) have been globally observed from the point of view of other threads in the system. This includes both read and write messages.

Data is called globally observable by other threads in the system when the data values written to the memory or data cache will now be returned by other threads' read messages when using that same data port. To read globally observable data that was written to a different data port, the thread issuing the data port read message needs to flush its cache (using a memory fence or pipe control) after the program knows that the writing thread issued the memory fence that ensured the global observability.

The memory fence message has an optional commit writeback message. The commit is sent only after all previous messages by this thread to that data port have been globally observed. This writeback can be used by threads to ensure that a fence is honored across both data ports, as each data port's memory fence only honors the corresponding data port messages.

Notes
BDW supports page faulting on some data cache operations. In the event of a page fault condition, the global observability of other data port operations may be impacted. For a thread to ensure that typed or untyped UAV accesses are visible to other threads, the memory fence message is used with the Commit Enable control to ensure that all page fault conditions in this thread have been handled. When the writeback message is returned to the thread, then any page fault conditions have been handled and the memory is globally observable to other threads. The normal usage is to issue a memory fence, source the writeback register, and then issue a gateway barrier message to release other threads to use the memory data. By using the writeback register from the fence message before issuing the next memory operation, the program guarantees that the fence has completed before the next data port message is issued.
The ordering and observability with SLM memory accesses is guaranteed between multiple threads in a thread group for the sequence of a write message, a barrier message, and then a read message.

Programming Note	
Context:	Memory Fence
<p>The writeback message returned with Commit Enable does not properly ensure that all page fault conditions in this thread have been handled. The workaround is to replace the memory fence message commit enable with the following two messages in this sequence:</p> <ul style="list-style-type: none"> Memory Fence message <u>without</u> Commit Enable Read Surface Info message (which returns data in a writeback message) <p>After the Read Surface Info writeback message is received by the thread, then all page fault conditions in this thread have been handled.</p>	

Message Descriptor

Bits	Description
13	Commit Enable Specifies whether the commit is returned to the thread after the fence has been honored. Format = Enable
12	L3_Flush_RW_Data If enabled causes the L3 to flush any RW data. If disabled RW data is not flushed.
11	L3_Flush_Constant_Data If enabled causes the L3 to flush any Constant data. If disabled Constant data is not flushed.
10	L3_Flush_Texture_Data If enabled causes the L3 to flush any Texture data. If disabled Texture data is not flushed.
9	L3_Flush_Instructions If enabled causes the L3 to flush any Instructions. If disabled Instructions are not flushed.
8	Reserved: MBZ

Programming Note	
Context:	Message Descriptor
Only one of the Flush controls (bits 12:9) may be specified per message.	

Programming Note	
Context:	Message Descriptor
The Flush controls (bits 12:9) are ignored.	

Programming Note	
Context:	Message Descriptor
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 then the entire partition is flushed.	

Programming Note	
Context:	Message Descriptor
The flushing of L3 is normally not needed. It is provided primarily to support workarounds, if needed.	

Programming Note	
Context:	Message Descriptor
SFID_DP_DC0 & SFID_DP_DC1 (unless forced non-coherent) are normally IA-coherent. Therefore no additional actions are needed by SW to ensure coherence. However, SW must use the memory fence (descriptor bits [12:8] can be all 0) to ensure that all memory cycles are visible throughout the memory hierarchy.	

Message Header

The fence messages consist of a single phase, and is completely ignored. The message length is always one.

DWord	Bits	Description
M0.7:0	31:0	Ignored

Writeback Message

The writeback message is only sent if **Commit Enable** in the message descriptor is set. The destination register is not modified. Memory fence messages without **Commit Enable** set do not return anything to the thread (response length is 0 and destination register is null).

DWord	Bits	Description
W0.7:0		Reserved

Pixel Data Port

Cache Agents

The data port allows access to memory via various caches. The choice of which cache to use for a given application is dictated by its restrictions, coherency issues, and how heavily that cache is used for other purposes.

The cache to use is selected by the shared function accessed.

Accessing Render Targets

Render targets are the surfaces that the final results of pixel shaders are written to. The render targets support a large set of surface formats (refer to surface formats table in *Sampling Engine* for details) with hardware conversion from the format delivered by the thread. The render target message also causes numerous side effects, including potentially alpha test, depth test, stencil test, alpha blend (which normally causes a read of the render target), and other functions. These functions are covered in the *Windower* chapter as some of them (depth/stencil test) are also partially done in the Windower.

The render target write messages are specifically for the use of pixel shader threads that are spawned by the windower, and may not be used by any other threads. This is due to the pixel scoreboard side-effects that sending of this message entails. The pixel scoreboard ensures that incorrect ordering of reads and writes to the same pixel does not occur.

Programming Note	
Context:	Accessing Render Targets
Half Precision Render Target Write messages do not support UNIT formats.	

Message Sequencing Summary

This section summarizes the sequencing that occurs for each legal render target write message. All messages have the M0 and M1 header registers if the header is present. If the header is not present, all registers below are renumbered starting with M0 where M2 appears. All cases not shown in this table are illegal.

Key:

s0, s1 = source 0, source 1

1/0 = slots 15:8

3/2 = slots 7:0

sZ = source depth

oM = oMask

Message Type	oMask Present	Source Depth Present	Source 0 Alpha Present	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
000	0	0	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A					
000	0	0	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A			
000	0	1	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ			
000	0	1	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ	
000	1	0	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A				
000	1	0	1	1/0s0A	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A		
000	1	1	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ		
000	1	1	1	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
001	0	0	0	RGBA												
001	1	0	0	oM	RGBA											
010	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
010	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
010	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
010	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
011	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
011	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
011	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
011	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
100	0	0	0	R	G	B	A									
100	0	0	1	s0A	R	G	B	A								
100	0	1	0	R	G	B	A	sZ								
100	0	1	1	s0A	R	G	B	A	sZ							
100	1	0	0	oM	R	G	B	A								
100	1	0	1	s0A	oM	R	G	B	A							
100	1	1	0	oM	R	G	B	A	sZ							
100	1	1	1	s0A	oM	R	G	B	A	sZ						

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 color (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.

The single source message does not cause a write to the render target if **Dual Source Blend Enable** in 3DSTATE_WM is *enabled*. 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.

Each dual source message delivered clears the corresponding pixel scoreboard bits if the **Last Render Target Select** bit in the message descriptor is set.

The dual source message reverts to a single source message using source 0 if **Dual Source Blend Enable** in 3DSTATE_WM is disabled.

Replicate Data

The replicate data render target message is used for “fast clear” functionality in cases where the color data for each pixel is identical. 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. This message must target only tiled memory. Access of linear memory using this message type is UNDEFINED. The depth buffer can be cleared through the “early depth” function in conjunction with a pixel shader using this message. Refer to the *Windower* chapter for more details on the early depth function.

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.

Multiple Render Targets (MRT)

Multiple render targets are supported with the single source and replicate data messages. Each render target is accessed with a separate Render Target Write message, each with a different surface indicated (different binding table index). The depth buffer is written 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 any of these surface formats: YCRCB_SWAPUVY, YCRCB_SWAPUV, YCRCB_SWAPY, or YCRCB_NORMAL.

Render Target Read and Write

Render Target Write

This message takes four subspans of pixels for write 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.

Programming Note	
Context:	Render Target Read and Write
<ul style="list-style-type: none"> 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, the surface format is further restricted to the set supported as alpha blend render target. The last message sent to the render target by a thread must have the End Of Thread bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard. The stateless model cannot be used with this message (Binding Table Index cannot be 255). 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. The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG. This message cannot be used on a surface in field mode (Vertical Line Stride = 1). If multiple SIMD8 Dual Source messages are delivered by the pixel shader thread, each SIMD8_DUALSRC_LO message must be issued <i>before</i> the SIMD8_DUALSRC_HI message with the same Slot Group Select setting. 	

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, the surface format is further restricted to the set supported as alpha blend render target.

The last message sent to the render target by a thread must have the **End Of Thread** bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard.

The stateless model cannot be used with this message (**Binding Table Index** cannot be 255).

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.

The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG.

This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1).

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.

Project-Specific Restrictions

Programming Note	
Context:	Render Target Read and Write
Execution Mask. For messages without header, the execution mask for render target messages (sent as part of the channel enables on the obus sideband) is used to kill pixels.	

Out-of-Bounds Accesses. Accesses to pixels outside of the surface are dropped and do not modify memory. However, if the **Render Target Array Index** is out of bounds, it is set to zero and the surface write is not suppressed.

The following table indicates the surface formats supported by this message with project restrictions and whether each format supports Alpha Blend.

Surface Format Name	Alpha Blend?
R32G32B32A32_FLOAT	Yes
R32G32B32A32_SINT	No
R32G32B32A32_UINT	No
R16G16B16A16_UNORM	Yes
R16G16B16A16_SNORM	Yes
R16G16B16A16_SINT	No
R16G16B16A16_UINT	No
R16G16B16A16_FLOAT	Yes
R32G32_FLOAT	Yes
R32G32_SINT	No
R32G32_UINT	No
B8G8R8A8_UNORM	Yes
B8G8R8A8_UNORM_SRGB	Yes
R10G10B10A2_UNORM	Yes
R10G10B10A2_UINT	No
R8G8B8A8_UNORM	Yes
R8G8B8A8_UNORM_SRGB	Yes
R8G8B8A8_SNORM	Yes
R8G8B8A8_SINT	No
R8G8B8A8_UINT	No
R16G16_UNORM	Yes
R16G16_SNORM	Yes
R16G16_SINT	No
R16G16_UINT	No
R16G16_FLOAT	Yes

Surface Format Name	Alpha Blend?
B10G10R10A2_UNORM	Yes
B10G10R10A2_UNORM_SRGB	Yes
R11G11B10_FLOAT	Yes
R32_SINT	No
R32_UINT	No
R32_FLOAT	Yes
B8G8R8X8_UNORM	Yes
B8G8R8X8_UNORM_SRGB	Yes
B5G6R5_UNORM	Yes
B5G6R5_UNORM_SRGB	Yes
B5G5R5A1_UNORM	Yes
B5G5R5A1_UNORM_SRGB	Yes
B4G4R4A4_UNORM	Yes
B4G4R4A4_UNORM_SRGB	Yes
R8G8_UNORM	Yes
R8G8_SNORM	Yes
R8G8_SINT	No
R8G8_UINT	No
R16_UNORM	Yes
R16_SNORM	Yes
R16_SINT	No
R16_UINT	No
R16_FLOAT	Yes
B5G5R5X1_UNORM	Yes
B5G5R5X1_UNORM_SRGB	Yes
R8_UNORM	Yes
R8_SNORM	Yes
R8_SINT	No
R8_UINT	No
A8_UNORM	Yes
YCRCB_NORMAL	No
YCRCB_SWAPUVY	No
YCRCB_SWAPUV	No
YCRCB_SWAPY	No

SubspanPixel to Slot Mapping

The following table indicates the mapping of subspans, pixels, and samples to slots in the pixel shader dispatch depending on the number of samples and message size. This table applies to all devices.

NumSamples Support

NumSamples Support
NumSamples = 1X, 2X, or 4X is supported for all projects.
NumSamples = 8X is supported.

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)

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]
SIMD16	8X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]

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

Programming Note	
Context:	SubspanPixel to Slot Mapping
<ul style="list-style-type: none"> When SIMD32 or SIMD16 PS threads send render target writes with multiple SIMD8 and SIMD16 messages, the following must hold: All the slots (as described above) must have a corresponding render target write irrespective of the slot's validity. A slot is considered valid when at least one sample is enabled. For example, a SIMD16 PS thread must send two SIMD8 render target writes to cover all the slots. PS thread must send SIMD render target write messages with increasing slot numbers. For example, SIMD16 thread has Slot[15:0] and if two SIMD8 render target writes are used, the first SIMD8 render target write must send Slot[7:0] and the next one must send Slot[15:8]. 	

Message Descriptor

This section contains descriptors for the render target read and write functions.

Message Descriptor - Render Target Write

Message Header

The render target write message has a two-register message header.

Message Header

DWord	Bits	Description																																
M0.5	31:8	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 on thread completion.																																
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)																																
M0.3	31:0	Ignored																																
M0.2	31:0	<p>Pixel Mask. One bit per pixel indicating which pixels are lit, possibly impacted by kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer. This field is ignored by the read message, all pixels are always returned.</p> <p>The bits in this mask correspond to the pixels as follows:</p> <table><tr><td>0</td><td>1</td><td>4</td><td>5</td><td>16</td><td>17</td><td>20</td><td>21</td></tr><tr><td>2</td><td>3</td><td>6</td><td>7</td><td>18</td><td>19</td><td>22</td><td>23</td></tr><tr><td>8</td><td>9</td><td>12</td><td>13</td><td>24</td><td>25</td><td>28</td><td>29</td></tr><tr><td>10</td><td>11</td><td>14</td><td>15</td><td>26</td><td>27</td><td>30</td><td>31</td></tr></table>	0	1	4	5	16	17	20	21	2	3	6	7	18	19	22	23	8	9	12	13	24	25	28	29	10	11	14	15	26	27	30	31
0	1	4	5	16	17	20	21																											
2	3	6	7	18	19	22	23																											
8	9	12	13	24	25	28	29																											
10	11	14	15	26	27	30	31																											
M0.1	31:0	<p>Y offset. The Y offset of the upper left corner of the block into the surface. Must be 4-row aligned (Bits 1:0 MBZ). Format = S31</p>																																
M0.0	31:0	<p>X offset. The X offset of the upper left corner of the block into the surface. This is a pixel offset assuming a 32-bit pixel. Must be 8-pixel aligned (Bits 2:0 MBZ). Format = S31</p>																																

Writeback Message (Read)

A SIMD16 writeback message consists of up to 8 destination registers. If a channel/component is not present in the RT format, the corresponding GRF is filled with zeroes or 1.0 in float/int depending on whether RGB or Alpha are disabled.

DWord	Bits	Description
W0.7	31:0	Slot 7 Red. Specifies the value of the red channel for slot 7. Format = 32 bits raw data.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1.7	31:0	Slot 15 Red
W1.6	31:0	Slot 14 Red
W1.5	31:0	Slot 13 Red
W1.4	31:0	Slot 12 Red
W1.3	31:0	Slot 11 Red
W1.2	31:0	Slot 10 Red
W1.1	31:0	Slot 9 Red
W1.0	31:0	Slot 8 Red
W2		Slots 7:0 Green
W3		Slots 15:8 Green
W4		Slots 7:0 Blue
W5		Slots 15:8 Blue
W6		Slots 7:0 Alpha
W7		Slots 15:8 Alpha

A SIMD8 writeback message consists of up to 4 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted channel mask results in the destination register of the corresponding channel being filled with zeroes or 1.0 in float/int depending on whether RGB or Alpha are disabled.

DWord	Bits	Description
W0.7	31:0	Slot 7 Red. Specifies the value of the red channel for slot 7. Format = 32 bits raw data.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1		Slots 7:0 Green
W2		Slots 7:0 Blue
W3		Slots 7:0 Alpha

Header for SIMD8_IMAGE_WRITE

DWord	Bits	Description													
M0.5	31:10	Ignored													
	9:8	Color Code: This ID is assigned by the Windower unit and is used to track synchronizng events. Format: Reserved for HW Implementation Use.													
	7:0	FFTID. The Fixed Function Thread 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.													
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)													
M0.3	31:0	Ignored													
M0.2	31:3	Ignored													
	2:0	Render Target Index. Specifies the render target index that will be used to select blend state from BLEND_STATE. Format = U3													
M0.1	31:6	ColorCalculatorState Pointer. Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:6] For SIMD8_IMAGE_WR message under normal GPGPU usage model, we recommend that SW program a dummy color-calc state such that all operations controlled by this state are disabled.													
	5:0	Ignored													
M0.0	31	Ignored													
	30:27	Viewport Index. Specifies the index of the viewport currently being used. Format = U4 Range = [0,15] SIMD8_IMAGE_WR message type this field is ignored by hardware.													
	26:16	Render Target Array Index. Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the "z" or "r" coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] SURFTYPE_BUFFER: must be zero. <table border="1"><thead><tr><th>face</th><th>Render Target Array Index</th></tr></thead><tbody><tr><td>+x</td><td>0</td></tr><tr><td>-x</td><td>1</td></tr><tr><td>+y</td><td>2</td></tr><tr><td>-y</td><td>3</td></tr><tr><td>+z</td><td>4</td></tr><tr><td>-z</td><td>5</td></tr></tbody></table> Format = U11 The Render Target Array Index used by hardware for access to the Render Target is overridden with the Minimum Array Element defined in SURFACE_STATE if it is out of the range between	face	Render Target Array Index	+x	0	-x	1	+y	2	-y	3	+z	4	-z
face	Render Target Array Index														
+x	0														
-x	1														
+y	2														
-y	3														
+z	4														
-z	5														

DWord	Bits	Description
		Minimum Array Element and Depth. For cube surfaces, a depth value of 5 is used for this determination. For SMD8_IMAGE_WRITE: For SURFTYPE_2D, this field must be 0. For SURFTYPE_3D, this field may not be 0 for "Write-3D-Image" operation.
	15:8	Ignored
	7:0	Pixel Masks for SIMD8 messages. 1: Pixel is enabled. 0: Pixel is disabled. In this case the corresponding (x,y) should be ignored by hardware.
M1.7	31:16	Y7: y-coordinate for pixel 7 Format = U16
	15:0	X7: x-coordinate for pixel 7 Format = U16
M1.6	31:16	Y6: y-coordinate for pixel 6 Format = U16
	15:0	X6: x-coordinate for pixel 6 Format = U16
M1.5	31:16	Y5: y-coordinate for pixel 5 Format = U16
	15:0	X5: x-coordinate for pixel 5 Format = U16
M1.4	31:16	Y4: y-coordinate for pixel 4 Format = U16
	15:0	X4: x-coordinate for pixel 4 Format = U16
M1.3	31:16	Y3: y-coordinate for pixel 3 Format = U16
	15:0	X3: x-coordinate for pixel 3 Format = U16
M1.2	31:16	Y2: y-coordinate for pixel 2 Format = U16
	15:0	X2: x-coordinate for pixel 2 Format = U16
M1.1	31:16	Y1: y-coordinate for pixel 1 Format = U16
	15:0	X1: x-coordinate for pixel 1 Format = U16
M1.0	31:16	Y0: y-coordinate for pixel 0 Format = U16
	15:0	X0: x-coordinate for pixel 0 Format = U16

Source 0 Alpha Payload

The source 0 alpha registers, if included, appear in M2 and M3, immediately following the header (if present).

For the SIMD8 single source message, only slot 7:0 data is sent (M2). The source 0 alpha phases are not supported for dual source messages.

DWord	Bit	Description
M2.7	31:0	Source 0 Alpha for Slot 7 Format = IEEE_Float This and the next register is only included if Source 0 Alpha Present bit is set.
M2.6	31:0	Source 0 Alpha for Slot 6
M2.5	31:0	Source 0 Alpha for Slot 5
M2.4	31:0	Source 0 Alpha for Slot 4
M2.3	31:0	Source 0 Alpha for Slot 3
M2.2	31:0	Source 0 Alpha for Slot 2
M2.1	31:0	Source 0 Alpha for Slot 1
M2.0	31:0	Source 0 Alpha for Slot 0
M3.7	31:0	Source 0 Alpha for Slot 15
M3.6	31:0	Source 0 Alpha for Slot 14
M3.5	31:0	Source 0 Alpha for Slot 13
M3.4	31:0	Source 0 Alpha for Slot 12
M3.3	31:0	Source 0 Alpha for Slot 11
M3.2	31:0	Source 0 Alpha for Slot 10
M3.1	31:0	Source 0 Alpha for Slot 9
M3.0	31:0	Source 0 Alpha for Slot 8

oMask Payload

The oMask payload, if present, follows source 0 alpha. The value of 'p' depends on whether the header and source 0 alpha are present.

Sample "n" for that pixel will be killed (not written to the render target or depth buffer) if bit "n" of the oMask is zero. Bits numbers where "n" is larger than the number of multisamples are ignored.

For the SIMD8 messages, only slots 7:0 data is used, or only slots 15:8 depending on the **Message Type** encoding.

DWord	Bit	Description
Mp.7	31:16	oMask for Slot 15 Format = 16-bit mask This register is only included if oMask Present bit is set.
	15:0	oMask for Slot 14
Mp.6	31:16	oMask for Slot 13
	15:0	oMask for Slot 12
Mp.5	31:16	oMask for Slot 11
	15:0	oMask for Slot 10
Mp.4	31:16	oMask for Slot 9
	15:0	oMask for Slot 8
Mp.3	31:16	oMask for Slot 7
	15:0	oMask for Slot 6
Mp.2	31:16	oMask for Slot 5
	15:0	oMask for Slot 4
Mp.1	31:16	oMask for Slot 3
	15:0	oMask for Slot 2
Mp.0	31:16	oMask for Slot 1
	15:0	oMask for Slot 0

Color Payload SIMD16 Single Source

Color Payload

This payload is included if the Message Type is SIMD16 single source. The value of 'm' depends on whether the header, source 0 alpha, and oMask are present.

DWord	Bits	Description
Mm.7	31:0	Slot 7 Red. Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	Slot 6 Red
Mm.5	31:0	Slot 5 Red
Mm.4	31:0	Slot 4 Red
Mm.3	31:0	Slot 3 Red
Mm.2	31:0	Slot 2 Red
Mm.1	31:0	Slot 1 Red
Mm.0	31:0	Slot 0 Red
M(m+1).7	31:0	Slot 15 Red
M(m+1).6	31:0	Slot 14 Red
M(m+1).5	31:0	Slot 13 Red
M(m+1).4	31:0	Slot 12 Red
M(m+1).3	31:0	Slot 11 Red
M(m+1).2	31:0	Slot 10 Red
M(m+1).1	31:0	Slot 9 Red
M(m+1).0	31:0	Slot 8 Red
M(m+2)		Slot[7:0] Green. See Mm definition for slot locations.
M(m+3)		Slot[15:8] Green. See M(m+1) definition for slot locations.
M(m+4)		Slot[7:0] Blue. See Mm definition for slot locations.
M(m+5)		Slot[15:8] Blue. See M(m+1) definition for slot locations.
M(m+6)		Slot[7:0] Alpha. See Mm definition for slot locations.
M(m+7)		Slot[15:8] Alpha. See M(m+1) definition for slot locations.

Color Payload SIMD8 Single Source

This payload is included if the Message Type is SIMD8 single source or SIMD8 Image Write. The value of 'm' depends on whether the header, source 0 alpha, and oMask are present.

DWord	Bits	Description
Mm.7	31:0	Slot 7 Red. Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	Slot 6 Red
Mm.5	31:0	Slot 5 Red
Mm.4	31:0	Slot 4 Red
Mm.3	31:0	Slot 3 Red
Mm.2	31:0	Slot 2 Red
Mm.1	31:0	Slot 1 Red
Mm.0	31:0	Slot 0 Red
M(m+1)		Slot[7:0] Green. See Mm definition for slot locations.
M(m+2)		Slot[7:0] Blue. See Mm definition for slot locations.
M(m+3)		Slot[7:0] Alpha. See Mm definition for slot locations.

Color Payload SIMD16 Replicated Data

This payload is included if the Message Type specifies a 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.

This message is legal with color data; oMask is also legal with this message. 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 value of 'm' depends on whether the header and oMask are present.

Note: This message is allowed only on tiled surfaces.

DWord	Bits	Description
Mm.7:4	31:0	Reserved
Mm.3	31:0	Alpha. Specifies the value of the alpha channel for all slots. Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.2	31:0	Blue
Mm.1	31:0	Green
Mm.0	31:0	Red

Color Payload SIMD8 Dual Source

This payload is included if the **Message Type** specifies dual source message. The value of 'm' depends on whether the header, source 0 alpha, and oMask are present.

Programming Note	
Context:	Color Payload SIMD8 Dual Source
The dual source message contains only 2 subspans (8 pixels) due to limitations in message length.	

DWord	Bits	Description
Mm.7	31:0	Slot 7 Source 0 Red. Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the Surface Format of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	Slot 6 Source 0 Red
Mm.5	31:0	Slot 5 Source 0 Red
Mm.4	31:0	Slot 4 Source 0 Red
Mm.3	31:0	Slot 3 Source 0 Red
Mm.2	31:0	Slot 2 Source 0 Red
Mm.1	31:0	Slot 1 Source 0 Red
Mm.0	31:0	Slot 0 Source 0 Red
M(m+1)		Slot[7:0] Source 0 Green. See Mm definition for slot locations.
M(m+2)		Slot[7:0] Source 0 Blue. See Mm definition for slot locations.
M(m+3)		Slot[7:0] Source 0 Alpha. See Mm definition for slot locations.
M(m+4)		Slot[7:0] Source 1 Red. See Mm definition for slot locations.
M(m+5)		Slot[7:0] Source 1 Green. See Mm definition for slot locations.
M(m+6)		Slot[7:0] Source 1 Blue. See Mm definition for slot locations.
M(m+7)		Slot[7:0] Source 1 Alpha. See Mm definition for slot locations.

Render Target Read and Write

Render Target Write

This message takes four subspans of pixels for write 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.

Programming Note	
Context:	Render Target Read and Write
<ul style="list-style-type: none"> 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, the surface format is further restricted to the set supported as alpha blend render target. The last message sent to the render target by a thread must have the End Of Thread bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard. The stateless model cannot be used with this message (Binding Table Index cannot be 255). 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. The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG. This message cannot be used on a surface in field mode (Vertical Line Stride = 1). If multiple SIMD8 Dual Source messages are delivered by the pixel shader thread, each SIMD8_DUALSRC_LO message must be issued <i>before</i> the SIMD8_DUALSRC_HI message with the same Slot Group Select setting. 	

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, the surface format is further restricted to the set supported as alpha blend render target.

The last message sent to the render target by a thread must have the **End Of Thread** bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard.

The stateless model cannot be used with this message (**Binding Table Index** cannot be 255).

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.

The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG.

This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1).

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.

Project-Specific Restrictions

Programming Note	
Context:	Render Target Read and Write
Execution Mask. For messages without header, the execution mask for render target messages (sent as part of the channel enables on the obus sideband) is used to kill pixels.	

Out-of-Bounds Accesses. Accesses to pixels outside of the surface are dropped and do not modify memory. However, if the **Render Target Array Index** is out of bounds, it is set to zero and the surface write is not suppressed.

The following table indicates the surface formats supported by this message with project restrictions and whether each format supports Alpha Blend.

Surface Format Name	Alpha Blend?
R32G32B32A32_FLOAT	Yes
R32G32B32A32_SINT	No
R32G32B32A32_UINT	No
R16G16B16A16_UNORM	Yes
R16G16B16A16_SNORM	Yes
R16G16B16A16_SINT	No
R16G16B16A16_UINT	No
R16G16B16A16_FLOAT	Yes
R32G32_FLOAT	Yes
R32G32_SINT	No
R32G32_UINT	No
B8G8R8A8_UNORM	Yes

Surface Format Name	Alpha Blend?
B8G8R8A8_UNORM_SRGB	Yes
R10G10B10A2_UNORM	Yes
R10G10B10A2_UINT	No
R8G8B8A8_UNORM	Yes
R8G8B8A8_UNORM_SRGB	Yes
R8G8B8A8_SNORM	Yes
R8G8B8A8_SINT	No
R8G8B8A8_UINT	No
R16G16_UNORM	Yes
R16G16_SNORM	Yes
R16G16_SINT	No
R16G16_UINT	No
R16G16_FLOAT	Yes
B10G10R10A2_UNORM	Yes
B10G10R10A2_UNORM_SRGB	Yes
R11G11B10_FLOAT	Yes
R32_SINT	No
R32_UINT	No
R32_FLOAT	Yes
B8G8R8X8_UNORM	Yes
B8G8R8X8_UNORM_SRGB	Yes
B5G6R5_UNORM	Yes
B5G6R5_UNORM_SRGB	Yes
B5G5R5A1_UNORM	Yes
B5G5R5A1_UNORM_SRGB	Yes
B4G4R4A4_UNORM	Yes
B4G4R4A4_UNORM_SRGB	Yes
R8G8_UNORM	Yes
R8G8_SNORM	Yes
R8G8_SINT	No
R8G8_UINT	No
R16_UNORM	Yes
R16_SNORM	Yes
R16_SINT	No
R16_UINT	No
R16_FLOAT	Yes

Surface Format Name	Alpha Blend?
B5G5R5X1_UNORM	Yes
B5G5R5X1_UNORM_SRGB	Yes
R8_UNORM	Yes
R8_SNORM	Yes
R8_SINT	No
R8_UINT	No
A8_UNORM	Yes
YCRCB_NORMAL	No
YCRCB_SWAPUVY	No
YCRCB_SWAPUV	No
YCRCB_SWAPY	No

Message Header

DWord	Bits	Description																																
M0.5	31:8	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 on thread completion.																																
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)																																
M0.3	31:0	Ignored																																
M0.2	31:0	<p>Pixel Mask. One bit per pixel indicating which pixels are lit, possibly impacted by kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer. This field is ignored by the read message, all pixels are always returned.</p> <p>The bits in this mask correspond to the pixels as follows:</p> <table><tr><td>0</td><td>1</td><td>4</td><td>5</td><td>16</td><td>17</td><td>20</td><td>21</td></tr><tr><td>2</td><td>3</td><td>6</td><td>7</td><td>18</td><td>19</td><td>22</td><td>23</td></tr><tr><td>8</td><td>9</td><td>12</td><td>13</td><td>24</td><td>25</td><td>28</td><td>29</td></tr><tr><td>10</td><td>11</td><td>14</td><td>15</td><td>26</td><td>27</td><td>30</td><td>31</td></tr></table>	0	1	4	5	16	17	20	21	2	3	6	7	18	19	22	23	8	9	12	13	24	25	28	29	10	11	14	15	26	27	30	31
0	1	4	5	16	17	20	21																											
2	3	6	7	18	19	22	23																											
8	9	12	13	24	25	28	29																											
10	11	14	15	26	27	30	31																											
M0.1	31:0	<p>Y offset. The Y offset of the upper left corner of the block into the surface. Must be 4-row aligned (Bits 1:0 MBZ). Format = S31</p>																																
M0.0	31:0	<p>X offset. The X offset of the upper left corner of the block into the surface. This is a pixel offset assuming a 32-bit pixel. Must be 8-pixel aligned (Bits 2:0 MBZ). Format = S31</p>																																

Shared Functions Pixel Interpolator

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

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

Messages

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

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

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

Initiating Message

Message Descriptor

Bits	Description
19	Header Present: Specifies whether the message includes a header phase. Must be zero for all <i>Pixel Interpolator</i> messages. Format = Enable
18:17	Ignored
16	SIMD Mode. Specifies the SIMD mode of the message being sent. Format = U1 0: SIMD8 mode 1: SIMD16 mode
15	Ignored
14	Interpolation Mode. Specifies which interpolation mode is used. Format = U1 0: Perspective Interpolation 1: Linear Interpolation Programming Note: This field cannot be set to “Linear Interpolation” unless Non-Perspective Barycentric Enable in 3DSTATE_CLIP is enabled.
13:12	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)

Bits	Description
	1: Sample Position Offset (eval_sindex) 2: Centroid Position Offset (eval_centroid) 3: Per Slot Offset (eval_snapped with register offset) Programming Note: 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.
11	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. Programming Note: This field must be set to SLOTGRP_LO for SIMD8 messages. SIMD8 messages always use bypassed data for slots 7:0.
10:8	Ignored
7:0	Message Specific Control. Refer to the sections below for the definition of these bits based on Message Type .

“Per Message Offset” Message Descriptor

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

“Sample Position Offset” Message Descriptor

Bits	Description
7:4	Sample Index Specifies the sample index that applies to all slots. : Sample Index must not exceed the value of NUM_RASTSAMPLES when NUM_RASTSAMPLES > 1. From API, perspective, Forced Sample Count Defines the maximum allowable index in this message. Format = U4 <div style="border: 1px solid black; padding: 2px; display: inline-block;"> Range [0, 15] </div>
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
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
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

DWord	Bit	Description
M3.3	31:0	Slot 11 Y Pixel Offset
M3.2	31:0	Slot 10 Y Pixel Offset
M3.1	31:0	Slot 9 Y Pixel Offset
M3.0	31:0	Slot 8 Y Pixel Offset

Writeback Message

SIMD8

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

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

SIMD16

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

DWord	Bit	Description
W0.7	31:0	Barycentric[1] for Slot 7 Format = IEEE_Float
W0.6	31:0	Barycentric[1] for Slot 6
W0.5	31:0	Barycentric[1] for Slot 5
W0.4	31:0	Barycentric[1] for Slot 4
W0.3	31:0	Barycentric[1] for Slot 3
W0.2	31:0	Barycentric[1] for Slot 2
W0.1	31:0	Barycentric[1] for Slot 1
W0.0	31:0	Barycentric[1] for Slot 0
W1.7	31:0	Barycentric[2] for Slot Format = IEEE_Float
W1.6	31:0	Barycentric[2] for Slot 6
W1.5	31:0	Barycentric[2] for Slot 5
W1.4	31:0	Barycentric[2] for Slot 4
W1.3	31:0	Barycentric[2] for Slot 3
W1.2	31:0	Barycentric[2] for Slot 2
W1.1	31:0	Barycentric[2] for Slot 1
W1.0	31:0	Barycentric[2] for Slot 0 Format = IEEE_Float
W2.7	31:0	Barycentric[1] for Slot 15
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.

FF_SYNC Messages

FF_SYNC messages pass critical information between GS/Clip threads and the GS/Clip FF units, as well as providing GS/Clip thread synchronization (ordering). GS threads report various counts resulting from running the GS and/or SO functions, prior to performing any output (to SOB buffers or to URB handles). Clip threads report only the number of handles required. A message response (writeback) length of 1 GRF is indicated on the 'send' instruction if the thread requires response data and/or synchronization. Refer to the GS/Clip stage chapter for details.

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
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: <ul style="list-style-type: none"> This bit must be 0 for URB_ATOMIC_* messages.
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: <ul style="list-style-type: none"> 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 <i>n</i> , 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.

Bit	Description
	Range = [0, 2047] for URB_ATOMIC* messages.
3:0	<p>URB Opcode</p> <p>0: URB_WRITE_HWORD</p> <p>1: URB_WRITE_OWORD</p> <p>2: URB_READ_HWORD</p> <p>3: URB_READ_OWORD</p> <p>4: URB_ATOMIC_MOV</p> <p>5: URB_ATOMIC_INC</p> <p>6: URB_ATOMIC_ADD</p> <p>7: URB_SIMD8_WRITE (see URB_SIMD8_* for descriptor details)</p> <p>8: URB_SIMD8_READ (see URB_SIMD8_* for descriptor details)</p> <p>9-15: Reserved</p>

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
M0.7	31:0	Debug
M0.6	31:0	Debug
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
	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

DWord	Bits	Description
	8	Vertex 0 DATA [0] Channel Mask
	7:0	Reserved
M0.4	31:0	<p>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.</p> <p>For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units.</p> <p>Format = U32</p> <p>Range = [0, 1023] for URB_*_HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined.</p> <p>Range = [0, 2047] for URB_*_OWORD messages. The range of the calculated offset must fall within the range [0, 2047] or behavior is undefined.</p>
M0.3	31:0	<p>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.</p> <p>For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units.</p> <p>Format = U32</p> <p>Range = [0, 1023] for URB_*_HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined.</p> <p>Range = [0, 2047] for URB_*_OWORD messages. The range of the calculated offset must fall within the range [0, 2047] or behavior is undefined.</p>
	31:0	Reserved.
M0.1	31:16	Handle ID 1. This ID is assigned by the fixed function unit and links the work in channel 1 to a specific entry within the fixed function unit. This field is ignored unless Swizzle Control indicates Interleave mode.
	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	Handle ID 0. This ID is assigned by the fixed function unit and links the work in channel 0 to a specific entry within the fixed function unit. This field is ignored unless Swizzle Control indicates Interleave mode.
	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

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 <u>will use</u> (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]
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
<ul style="list-style-type: none"> The URB function <u>will use</u> (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]
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.

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]
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]
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]
...		...
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 <u>will use</u> (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

DWord	Bit	Description
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
<ul style="list-style-type: none"> The URB function <u>will use</u> (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]
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	Project
URB_ATOMIC_MOV	0	none	
URB_ATOMIC_INC	old_dst + 1	old_dst	
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	Security
M0.7	31:0	Debug	
M0.6	31:0	Debug	
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 <i>n</i> , 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 6: URB_ATOMIC_ADD 7: URB_SIMD8_WRITE 8: URB_SIMD8_READ 9-15: Reserved

Message Header

DWord	Bits	Description
M0.7	31:16	Handle ID 7. This ID is assigned by the fixed function unit and links the work in channel 7 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 7. This is the URB handle where channel 7's results are written or read.
M0.6	31:16	Handle ID 6. This ID is assigned by the fixed function unit and links the work in channel 6 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 6. This is the URB handle where channel 6's results are written or read.
M0.5	31:16	Handle ID 5. This ID is assigned by the fixed function unit and links the work in channel 5 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 5. This is the URB handle where channel 5's results are written or read.
M0.4	31:16	Handle ID 4. This ID is assigned by the fixed function unit and links the work in channel 4 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 4. This is the URB handle where channel 4's results are written or read.
M0.3	31:16	Handle ID 3. This ID is assigned by the fixed function unit and links the work in channel 3 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 3. This is the URB handle where channel 3's results are written or read.
M0.2	31:16	Handle ID 2. This ID is assigned by the fixed function unit and links the work in channel 2 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 2. This is the URB handle where channel 2's results are written or read.
M0.1	31:16	Handle ID 1. This ID is assigned by the fixed function unit and links the work in channel 1 to a specific entry within the fixed function unit.
	15:14	Reserved
	13:0	URB Handle 1. This is the URB handle where channel 1's results are written or read.
M0.0	31:16	Handle ID 0. This ID is assigned by the fixed function unit and links the work in channel 0 to a specific entry within the fixed function unit.
	15:14	Reserved
	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 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
	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

DWord	Bit	Description
	15:0	Reserved.
M2.0	31:24	Reserved.
	23:16	Vertex 0 DATA [7:0] Channel Mask
	15:0	Reserved.

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]
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]
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 shared function provides a mechanism for active thread-to-thread communication. Such thread-to-thread communication is based on direct register access. One thread, a *requester thread*, is capable of writing into the GRF register space of another thread, a *recipient thread*. Such direct register access between two threads in a multi-processor environment some time is referred to as *remote register access*. Remote register access may include read or write. The architecture supports *remote register write*, but not remote register read (natively). Message Gateway facilitates such remote register write via message passing. The requester thread sends a message to Message Gateway requesting a write to the recipient thread's GRF register space. Message Gateway sends a writeback message to the recipient thread to complete the register write on behalf of the requester. The requester thread and the recipient thread may be on the same EU or on different EUs.

Please see Thread Spawn Message Section of Media Chapter for child thread termination using Message Gateway messages with EOT bit set.

When Bypass Gateway Control is set to 1, the commands OpenGateway and CloseGateway are no longer used, the gateway parameters are taking the default values as the following:

- **RegBase** = 0
- **Gateway Size** check and **Key** check are bypassed.
- **Gateway Open** (an internal signal that is used to be set by OpenGateway message) check is bypassed

A separate Gateway exists per half-slice in the architecture. For ForwardMsg this is handled transparently, but barriers can only be accessed by threads in the local half-slice. This means that all threads that access a shared barrier need to use the half-slice select in GPGPU_OBJECT and MEDIA_OBJECT to stay on a single half-slice. GPGPU_WALKER handles this automatically.

Messages

Message Gateway supports such thread-to-thread communication with the following messages:

- **OpenGateway:** Opens a gateway for a requester thread. Once a thread successfully opens its gateway, it can be a recipient thread to receive remote register write.
- **CloseGateway:** Closes the gateway for a requester thread. Once a thread successfully closes its gateway, Message Gateway blocks any future remote register writes to this thread.
- **ForwardMsg:** Forwards a formatted message (remote register write) from a requester thread to a recipient thread.
- **GetTimeStamp:** Reads absolute and relative timestamps for a requester thread.
- **BarrierMsg:** A set of threads sends this message to the Gateway. When all threads in a group have sent the message, a reply (both a register write and an NO notification) is sent to each member of the group.
- **UpdateGatewayState:** Updates the internal state of the Message Gateway.

One example usage is to allow a control thread to change Barrier Byte to convey dynamic state information. This may be used to support interrupt when persistent compute/worker threads are synchronized using Barrier.

Message Descriptor

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

Bits	Description
19	Header Present. This bit must be 0 for all Message Gateway messages.
18:17	Ignored.
16:15	Notify. Send Notification Signal. This is a two-bit field indicating which notify event is sent. 00b: No notify. 01b: Increment recipient thread's N0 notification counter. 10b: Increment recipient thread's N2 notification counter. 11b: Reserved. This field is only valid for a ForwardMsg message. It is ignored for other messages. The BarrierMsg message always increments the N0 notification counter.
14	AckReq. Acknowledgment Required. When this bit is set, an acknowledgment return message is required. Message Gateway sends a writeback message containing the error code to the requester thread using the post destination register address. When this bit is 0, no writeback message is sent to the requesting thread by Message Gateway, even if an error occurs. This field is valid for OpenGateway, CloseGateway, ForwardMsg, and BarrierMsg messages. When this bit is 1, post destination register must be valid and the response length must be 1. When this bit is 0, post destination register must be <i>null</i> and the response length must be 0. This bit cannot be set when EOT is set; otherwise, hardware behavior is undefined. 0: No Acknowledgement is required. 1: Acknowledgement is required.
13:3	Reserved: MBZ
2:0	SubFuncID. Identify the supported sub-functions by Message Gateway. Encodings are: 000b: OpenGateway. Open the gateway for the requester thread. 001b: CloseGateway. Close the gateway for the requester thread. 010b: ForwardMsg. Forward the formatted message to the recipient thread with the given offset from the recipient's register base. 011b: GetTimeStamp. Read absolute and relative timestamps. 100b: BarrierMsg. Record an additional thread reaching the barrier. 101b: UpdateGatewayState. Update the barrier byte for a barrier. 111b: Reserved.

OpenGateway Message

The OpenGateway message opens a communication channel between the requesting thread and other threads. It specifies a key for other threads to access its gateway, as well as the GRF register range allowed to be written. The message consists of a single 256-bit message payload.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completion of the OpenGateway function. Only the least significant DWord in the post destination register is overwritten.

If the EOT is set for this message, Message Gateway ignores this message; instead, it closes the gateway for the requesting thread regardless of the previous state of the gateway.

It is software's policy to determine how to generate the key.

The BarrierMsg command does not use an OpenGateway message.

Message Payload

DWord	Bits	Description
M0.5	31:29	Reserved: MBZ
	28:21	RegBase: The register base address to be stored in the Message Gateway. It is used to compute the destination GRF register address from the offset field in ForwardMsg. RegBase contains 256-bit GRF aligned register address. Note 1: This field aligns with bits [28:21] of the Offset field of the message payload for ForwardMsg. Note 2: The most significant bit of this field must be zero. Format = U8 Range = [0,127]
	20:11	Reserved: MBZ
	10:8	Gateway Size: The range limit for messages through the Message Gateway. 000b: 1 GRF Register 001b: 2 GRF Registers 010b: 4 GRF Registers 011b: 8 GRF Registers 100b: 16 GRF Registers 101b: 32 GRF Registers 110b: 64 GRF Registers 111b: 128 GRF Registers
	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	31:16	Reserved: MBZ
	15:0	Reserved: MBZ.
M0.3:0	31:0	Ignored

Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bits	Description
W0.7:1	31:0	Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	Shared Function ID. The message gateway's shared function ID.
	15:3	Reserved
	2:0	Error Code 000b: Successful. No Error (Normal). 101b: Opcode Error. Attempt to send a message which is not either open/close/forward. Other codes: Reserved.

CloseGateway Message

The CloseGateway message closes a communication channel for the requesting thread that was previously opened with OpenGateway. Each thread is allowed to have only one open gateway at a time, thus no additional information in the message payload is required to close the gateway. The message consists of a single 256-bit message payload.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completing the CloseGateway function. Only the least significant DWord in the post destination register is overwritten.

Programming Note	
Context:	CloseGateway Message
The BarrierMsg command does not use a CloseGateway message.	

Message Payload

DWord	Bit	Description
M0.7:6		Ignored
M0.5	31:8	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. 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.
M0.4:0		Ignored

Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	Shared Function ID: Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	Error Code 000: <i>Successful</i> . No Error (Normal) 101: <i>Opcode Error</i> . Attempt to send a message which is not either open/close/forward other codes: Reserved

ForwardMsg Message

The ForwardMsg message gives the ability for a requester thread to write a **data segment** in the form of a byte, a dword, 2 dwords, or 4 dwords to a GRF register in a recipient thread. The message consists of a single 256-bit message payload, which contains the specially formatted data segment.

The ForwardMsg message utilizes a communication channel previously opened by the recipient thread. The recipient thread has communicated its EUID, TID, and key to the requester thread previously via some other mechanism. Generally, this is done through the thread spawn message from parent to child thread, allowing each child (requester) to then communicate with its parent through a gateway opened by the parent (recipient). The child could then use ForwardMsg message to communicate its own EUID, TID, and key back to the parent to enable bi-directional communication after opening its own gateway.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requester thread after completion of the ForwardMsg function. Only the least significant DWord in the post destination register is overwritten.

If the Notify bit in the message descriptor is set, a 'notification' is sent to the recipient thread in order to increment the recipient thread's notification counter. This allows multiple messages to be sent to the recipient without waking up the recipient thread. The last message, having this bit set, will then wake up the recipient thread.

Message Payload

DWord	Bits	Description
M0.5	31:29	Reserved: MBZ
	28:16	Offset: It provides the destination register position in the recipient thread GRF register space as the offset from the RegBase stored in the recipient thread's gateway entry. The offset is in unit of byte, such that bits [28:21] is the 256-bit aligned register offset and bits [4:0] is the sub-register offset. The sub-register offset must be aligned to the Length field in bits [10:8]. The subfields of Offset are further illustrated as the following. Offset[28:21]: Register offset from the gateway base (Range [0, 127]: bit 12 MBZ) Offset[20:18]: DW offset Offset[17:16]: Byte offset (must be 00b for all DW length cases) Programming Restriction: R0 cannot be used as destination GRF register for ForwardMsg. NULL register is also not allowed as destination.
	15:11	Reserved: MBZ
	10:8	Length: The length of the data segment. 000: 1 byte 001: 1 word 010: 1 DWord 011: 2 DWords 100: 4 DWords 101-111: Reserved
	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.

DWord	Bits	Description
M0.4	31:30	Ignored
	31:30	SliceID: The SliceID identifies the slice of the message destination EU. The behavior of sending a message to a non-existent or disabled slice is undefined.
	29:28	SubSliceID: The sub-slice ID identifies the sub-slice containing the message destination EU. The behavior of sending a message to a non-existent or disabled slice is undefined.
	27:24	EUID: The Execution Unit ID as part of the Recipient field is used to identify the recipient thread to which the message is forwarded.
	23:19	Ignored
	18:16	TID: The Thread ID as part of the Recipient field is used to identify the recipient thread to which the message is forwarded.
	15:0	Key: The key to match with the one stored in the recipient thread's entry in Message Gateway. BDW: Ignored.
	28:26	SliceID: The Slice ID as part of the Recipient field is used to identify the slice containing the EU to which the message is forwarded.
	25:24	SubSliceID: The sub-slice ID identifies the sub-slice containing the message destination EU. The behavior of sending a message to a non-existent or disabled slice is undefined.
	23:20	EUID: The Execution Unit ID as part of the Recipient field is used to identify the recipient thread to which the message is forwarded.
	19	Ignored
	18:16	TID: The Thread ID as part of the Recipient field is used to identify the recipient thread to which the message is forwarded.
	15:0	Ignored
M0.3	31:0	Data Segment DWord 3: Valid only for the 4-DWord data segment length.
M0.2	31:0	Data Segment DWord 2: Valid only for the 4-DWord data segment length.
M0.1	31:0	Data Segment Dword 1: Valid only for the 2- and 4-DWord data segment lengths.
M0.0	31:24	Data Segment Byte 0: The same byte must be copied to all four positions within this DWord. Valid only for the 1-Byte data segment length. Data Segment Dword 0: Valid only for the 1-, 2-, and 4-DWord data segment lengths.
	23:16	Data Segment Byte 0
	15:8	Data Segment Byte 0
	7:0	Data Segment Byte 0

Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bits	Description
W0.7:1	31:0	Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	Shared Function ID. The message gateway's shared function ID.
	15:3	Reserved
	2:0	Error Code 000b: Successful. No Error (Normal). 001b: Reserved. 010b: Gateway Closed. Attempt to send a message through a closed gateway. 011b: Reserved. 100b: Reserved. 101b: Opcode Error. Attempt to send a message which is not either open/close/forward. 110b: Invalid Message Size. Attempt to forward a message with length greater than 4 DWords. 111b: Reserved.

Writeback Message to Recipient Thread

This message contains the byte or dwords data segment indicated in the message written to the GRF register offset indicated. Only the byte/dword(s) will be enabled, all other data in the GRF register is untouched.

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
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	Ignored
M0.2	31:30	Ignored
	29:28	Ignored
	27:24	BarrierID. This field indicates which one from the 16 Barrier States is updated. 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.
	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.

DWord	Bits	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	Shared Function ID. Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	Error Code 000: <i>Successful.</i> No Error (Normal). 001: <i>Error (Barrier is inactive)</i> Other encodings are reserved.

Broadcast Writeback Message

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.

DWord	Bits	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:16	Reserved (not overwritten)
	15:8	Reserved (not overwritten)
	7:0	Reserved (not overwritten)

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 **PartToleranceThrd** 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

Minor Shape Decision Prior to FME

If any minor shapes are selected, we decide the best minor first.

For each 8x8 sub-block, before performing bidirectional, we reduce code candidates to no more than three based on the best unidirectional motion search results (best of the forward and backward):

- 0) One MV, i.e. the best in shape of 8x8.
- 1) Up to two MVs, i.e. the best in shapes 8x8, 8x4, or 4x8. And
- 2) Up to four MVs, i.e. the best for the sub-block 8x8.

Now for the first and the second sub-blocks, we can merge them into up to six candidates of 2, 3, 4, 5, 6, and 8 possible motion vectors.

Do the same to the third and the fourth sub-blocks; we have similarly up to six candidates.

Now we further combine these two groups, and find the best solution under the constraint of not exceeding the number of motion vectors more than **MaxNumMVs** (see pseudo-code below for detail).

Consequently, we have the best combined 8x8 solutions with **N** motion vectors for some **N** less or equal to **MaxNumMVs**.

Assume `distA[k][s]` is the cost-adjusted distortion of the best forward or backward motion vector mix of the k -th 8x8 sub-block of the sub-shape s , where $s=0, 1, 2$, and 3 represent shape partitioning 8x8, 8x4, 4x8, and 4x4 respectively. Assume `distA[k][s]` is the bidirectional one of the corresponding sub-block and sub-shape. And assume some large number, say $128 \times 16 = 2048$ is assigned to the variable, if there were no valid corresponding codes. Hence, the following pseudo-code explains the code selection algorithm.

Let's first explain the case where **MaxNumMVs** is disabled, i.e. **MaxNumMVs** ≥ 16 :

```
void SelectBestCombinedMinors(
    short *distA,
    short *MinorShape,
    short *MinorDisto)
{
    short s[4], d[4];
    s = ShapeList;
    d = DistoList;
    for ( int k=0; k<4; k++ ) {
        s[k] = 0;
        d[k] = distA[k][0];
        if ( distA[k][1]<d[k] ) { d[k] = distA[k][1]; s[k] = 1; }
        if ( distA[k][2]<d[k] ) { d[k] = distA[k][2]; s[k] = 2; }
        if ( distA[k][3]<d[k] ) { d[k] = distA[k][3]; s[k] = 3; }
    }
    * MinorDisto = d[0] + d[1] + d[2] + d[3];
    * MinorShape = s[0] | (s[1]<<2) | (s[2]<<4) | ((s[3]<<6));
}
```

Now for the case of using **MaxNumMVs** control:

```
void SelectBestCombinedMinors(
    short *distA,
    int    MaxNumMVs,
    short *MinorShape,
```



```

short *MinorDisto)
{
    int k, n;
    short dist, best0 = 0, best1 = 0;
    if ( MaxNumMVs < 4 ) { // We reset other parameters.
        switch ( MaxNumMvs ) {
            case 0:
                DoIntraInter &= (~DO_INTER); // Not do Inter
                break;

            case 1:
                ShapeMask |= (NO_16X8 | NO_8X16);
                BidirMask |= NO_16X16;
                break;

            case 2:
            case 3:
                ShapeMask |= (NO_8X8 | NO_8X4 | NO_4X8 | NO_4X4);
                BidirMask |= (NO_16X8 | NO_8X16);
                break;
        }
    }
    if ( MaxNumMVs >= 16 ) { // It should use unrestricted code selection.
        SelectBestCombinedMinors(DistA, MinorShape, MinorDisto);
        return;
    }
    short *s, ShapeList[18];
    short *d, DistoList[18];
    s = ShapeList;
    d = DistoList;
    for ( k=0; k<4; k++ ){
        s[0] = 0; // 1 mv
        d[0] = distA[k][0];
        s[4] = (distA[k][2] < distA[k][1]) + 1; // 2 mvs
        d[4] = distA[k][s[1]];
        s[8] = 3; // 4 mvs
        d[8] = distA[k][3];
        s ++, d ++;
    }
    // Merge two:
    s = ShapeList;
    d = DistoList;
    for ( k=0; k<2; k++ ) {
        s[16] = 0x33; // 8 mvs
        d[16] = d[8] + d[10];

        s[12] = (d[4] + d[10] < d[6] + d[8]) ? (s[4] | 0x30) : (0x03 | (s[6] << 4)); // 6
mvs
        d[12] = (d[4] + d[10] < d[6] + d[8]) ? (d[4] + d[10]) < (d[6] + d[8]);

        s[10] = (d[0] + d[10] < d[8] + d[2]) ? 0x30 : 0x03; // 5 mvs
        d[10] = (d[0] + d[10] < d[8] + d[2]) ? (d[0] + d[10]) < (d[8] + d[2]);

        s[8] = s[4] | (s[6] << 4); // 4 mvs
        d[8] = d[4] + d[6];

        s[6] = (d[4] + d[2] < d[0] + d[6]) ? s[4] : (s[6] << 4); // 3 mvs
        d[6] = (d[4] + d[2] < d[0] + d[6]) ? (d[4] + d[2]) < (d[0] + d[6]);

        s[4] = 0; // 2 mvs
        d[4] = d[0] + d[2];

```

```

    if ( d[ 6] > d[ 4] ) d[ 6] = d[ 4];
    if ( d[ 8] > d[ 6] ) d[ 8] = d[ 6];
    if ( d[10] > d[ 8] ) d[10] = d[ 8];
    if ( d[12] > d[10] ) d[12] = d[10];
    d[14] = d[12];
    if ( d[16] > d[12] ) d[16] = d[12];

    s ++; d ++;
  }
  s = ShapeList;
  d = DistoList;
  * MinorDisto = 2048;
  for ( k=0; k<8; k++ ) {
    n = MaxNumMVs - k;
    if ( (n>=2 && n<=8) <2 ) {
      dist = d[(k << 1) + 1] + d[n << 1];
      if ( dist < *MinorDisto ) {
        *MinorDisto = dist;
        best0 = (n << 1);
        best1 = (k << 1) + 1;
      }
    }
  }
  while ( best0 > 1 && d[best0] == d[best0-2] ) best0 -= 2;
  while ( best1 > 1 && d[best1] == d[best1-2] ) best1 -= 2;
  *MinorShape = s[best0] | (s[best1] << 2);
}

```

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 shapes**.

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 $\text{DistoMajor}[k] \leq \text{BestDisto} + \text{IntTol}$.

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

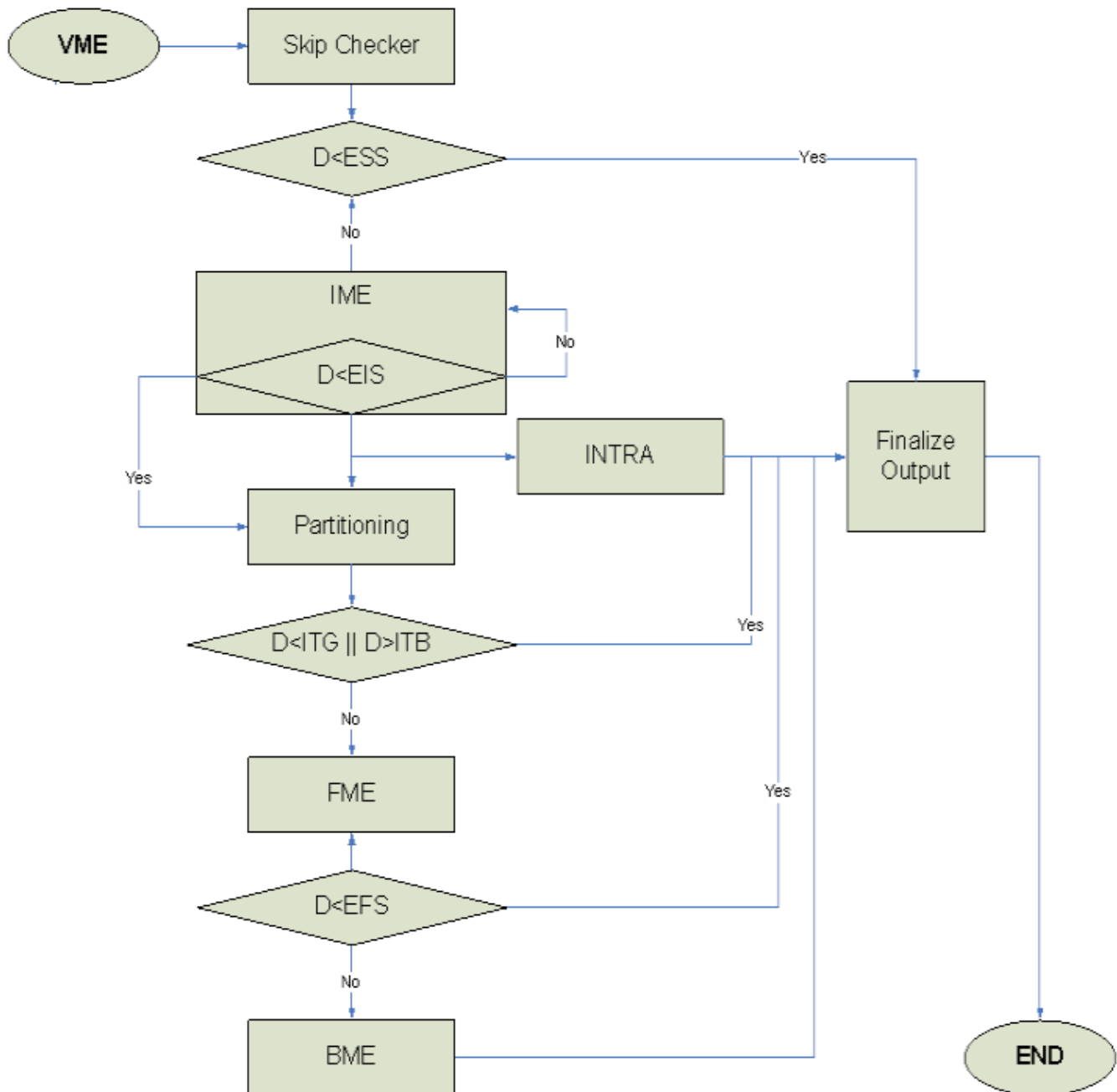
hits above **MaxNumMVs**; in this case, we take as many as possible first the ones generate the most improvement.

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

Early Decisions

There are 5 programmable early decision states are available for fine control of the VME process. All stored in one byte of U4U4 format to representing a value of (B«S), (where B, called **base**, is the 4-LSB of the byte and S, called **shift**, is the 4-MSB of the byte), they are the following:

1. ESS: EarlySkipSuccess = Early successful return after Skip is checked
2. EIS: EarlyImeStop = Early IME stop when a good match is found inside of IME process.
3. ITG: ImeTooGood = Early successful return after IME is done when a good enough match is found.
4. ITB: ImeTooBad = Early termination do skip fractional and bidirectional refinement after IME is done with a hopelessly bad match as the best result.
5. EFS: EarlyFmeSuccess = Early Success after Fractional ME to skip bidirectional search.



For any reason, if all possible code types are not chosen, VME will return Intra16x16 type with all modes set to 0, and the **MinDist** is set to 0x3FFF.

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 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)
	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)
	15:8	LUT_MbMode [1] for Set 0 Format = U4U4 (encoded value must fit in 12-bits)

DWord	Bit	Description
	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)
	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 RefIdx (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 $(0xA \ll 0x4) = 10 * 16 = 160$. 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 $dx = \text{int}(dx)$;

Costing_penalty_x = LUT_MV[p+1], else if $dx = 2^p$, for any $p \leq 6$;

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).

Software Interface – PRM Highlights

Message Structure Overview

- The contents of each message are different, but they have structural similarities to reduce coding complexity.
- The first 3 input phases (*'Message Phase' == 1 GRF of the message payload*) are structurally the same, given the mnemonic "Universal". Individual fields within the Universal phase are ignored based on message type.
- Additional input phases are appended to each message type to fulfill the required inputs only exclusive to that message type.
- Specifically, 4 message phases are appended to SIC (SIC0-SIC3), either 2, 4 or 6 message phases (based on streamin\streamout) are appended to IME (IME0-IME5), and 4 message phases are appended to FBR (FBR0-FBR3).
- The programmer is required to pack the necessary GRFs together to generate the correct message phase sequence before calling VME (i.e. 7 phases for SIC; 5, 7, or 9 phases for IME; and 7 phases for FBR).
- The return data is structurally common for all 3 message types, given in 7 phases. The only exception is IME return data when streamout data is present, then 2 or 4 additional phases are returned.
- Additionally, the placement of individual fields within the message phases is generally identical to that of previous generations.

IME and IDM Message Descriptor

Bits	Description
19	Header Present. If set, indicates that the message includes the header. This bit must be 1 for all VME messages. Format = Enable
18	Reserved: MBZ
17	Stream-Out Enable 2nd-Best. If set, additional message phases of record stream-out are present with the output of IME message containing the 2nd best candidates for each shape partition: 4 additional phases only when search control (M0.3 10:8) is 111b (dual reference & dual record) and 2 additional phases otherwise. Format = Enable
16	Stream-In Enable. If set, additional message phases of record stream-in are present with the input of IME message: 4 additional phases only when search control (M0.3 10:8) is 111b (dual reference & dual record) and 2 additional phases otherwise. Format = Enable
15	Stream-Out Enable. If set, additional message phases of record stream-out are present with the output of IME message: 4 additional phases only when search control (M0.3 10:8) is 111b (dual reference & dual record) and 2 additional phases otherwise. Format = Enable
14:13	Message Type 00: IDM 01: Reserved 10: IME 11: Reserved
12:8	Reserved: MBZ
7:0	Binding Table Index. Specifies the index into the binding table for the source surface. Format = U8 Range = [0,254]

Input GRFs

GRF	Name	Msgs	New	Major Contents
0	Uni0	ALL	No	Universal control data
1	Uni1	ALL	No	Universal control data
2	Uni2	ALL	No	Costs, FT Matrix, FBR Modes
3	Uni3	ALL	Yes	8 CostCenter MVs
4	SIC0	SIC	No	8 Skip MVs
5	SIC1	SIC	No	Luma intra pix, modes, masks
6	SIC2	SIC	No	Luma intra pix, modes, masks
7	SIC3	SIC	No	Chroma intra pix & masks
8	IME0	IME	No	Search Path
9	IME1	IME	No	Search Path
10	IME2	IME	No	Streamin\Streamout
11	IME3	IME	No	Streamin\Streamout
12	IME4	IME	No	Streamin\Streamout
13	IME5	IME	No	Streamin\Streamout
14	FBR0	FBR	No	8 Inter 4x4 MVs
15	FBR1	FBR	No	8 Inter 4x4 MVs
16	FBR2	FBR	No	8 Inter 4x4 MVs
17	FBR3	FBR	No	8 Inter 4x4 MVs
18	IDM0	IDM	Yes	Source Pixel Mask

Input Message Phases by Type

VME message types require only a subset of the total GRFs of control data.

Phase	SIC	IME	FBR	IDM
0	Uni0	Uni0	Uni0	Uni0
1	Uni1	Uni1	Uni1	Uni1
2	Uni2	Uni2	Uni2	Uni2
3	Uni3	Uni3	Uni3	Uni3
4	SIC0	IME0	FBR0	IDM0
5	SIC1	IME1	FBR1	
6	SIC2	IME2	FBR2	
7	SIC3	IME3	FBR3	
8		IME4		
9		IME5		

Output GRFs

GRF	Name	Msgs	New	Major Contents
0	Ret0	ALL	No	Best MB Control Data
1	Ret1	ALL	No	8 Inter 4x4 MVs
2	Ret2	ALL	No	8 Inter 4x4 MVs
3	Ret3	ALL	No	8 Inter 4x4 MVs
4	Ret4	ALL	No	8 Inter 4x4 MVs
5	Ret5	ALL	No	Inter Block Distortions
6	Ret6	ALL	No	Block Ref Indices & FTQ Data
7	IME2	IME	No	Streamin/Streamout
8	IME3	IME	No	Streamin/Streamout
9	IME4	IME	No	Streamin/Streamout
10	IME5	IME	No	Streamin/Streamout
11	IDM0	IDM	Yes	16 Search Point Dists
12	IDM1	IDM	Yes	16 Search Point Dists
13	IDM2	IDM	Yes	16 Search Point Dists
14	IDM3	IDM	Yes	16 Search Point Dists
15	IDM4	IDM	Yes	16 Search Point Dists
16	IDM5	IDM	Yes	16 Search Point Dists
17	IDM6	IDM	Yes	16 Search Point Dists
18	IDM7	IDM	Yes	16 Search Point Dists
19	IDM8	IDM	Yes	16 Search Point Dists
20	IDM9	IDM	Yes	16 Search Point Dists
21	IDM10	IDM	Yes	16 Search Point Dists
22	IDM11	IDM	Yes	16 Search Point Dists
23	IDM12	IDM	Yes	16 Search Point Dists
24	IDM13	IDM	Yes	16 Search Point Dists
25	IDM14	IDM	Yes	16 Search Point Dists
26	IDM15	IDM	Yes	16 Search Point Dists

Output Message Phases by Type

All message types return 7 phases except IDM. IME returns 2 or 4 additional phases of streamout if it is enabled (2 for uni, 4 for bi). Note the IME streamout message phases are structurally identical to the IME streamin phases. The IDM message will return only the 16 phases of distortion mesh output.

Phase	SIC	IME	FBR	IDM
0	Ret0	Ret0	Ret0	IDM0
1	Ret1	Ret1	Ret1	IDM1
2	Ret2	Ret2	Ret2	IDM2
3	Ret3	Ret3	Ret3	IDM3
4	Ret4	Ret4	Ret4	IDM4
5	Ret5	Ret5	Ret5	IDM5
6	Ret6	Ret6	Ret6	IDM6
7		IME2		IDM7
8		IME3		IDM8
9		IME4		IDM9
10		IME5		IDM10
11				IDM11
12				IDM12
13				IDM13
14				IDM14
15				IDM15

Binding Table Pointers

The following gives the driver and HW perspective of how the RefID will map to the binding table pointers indices (and hence surface state). The fixed mapping simplifies the HW definition.

Progressive Content

	Driver View			Universal Input M1.6 RefIDs (4b Value per Block)								Conversion
BTI	Direction	Number	Field	FWD 0	BWD 0	FWD 1	BWD 1	FWD 2	BWD 2	FWD 3	BWD 3	BTI Equation
0	Source	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	= From input
1	FWD	0	N/A	0	N/A	0	N/A	0	N/A	0	N/A	= RefID * 2 + 1
2	BWD	0	N/A	N/A	0	N/A	0	N/A	0	N/A	0	= RefID * 2 + 2
3	FWD	1	N/A	1	N/A	1	N/A	1	N/A	1	N/A	= RefID * 2 + 1
4	BWD	1	N/A	N/A	1	N/A	1	N/A	1	N/A	1	= RefID * 2 + 2
5	FWD	2	N/A	2	N/A	2	N/A	2	N/A	2	N/A	= RefID * 2 + 1
6	BWD	2	N/A	N/A	2	N/A	2	N/A	2	N/A	2	= RefID * 2 + 2
7	FWD	3	N/A	3	N/A	3	N/A	3	N/A	3	N/A	= RefID * 2 + 1
8	BWD	3	N/A	N/A	3	N/A	3	N/A	3	N/A	3	= RefID * 2 + 2
9	FWD	4	N/A	4	N/A	4	N/A	4	N/A	4	N/A	= RefID * 2 + 1
10	BWD	4	N/A	N/A	4	N/A	4	N/A	4	N/A	4	= RefID * 2 + 2
11	FWD	5	N/A	5	N/A	5	N/A	5	N/A	5	N/A	= RefID * 2 + 1
12	BWD	5	N/A	N/A	5	N/A	5	N/A	5	N/A	5	= RefID * 2 + 2
13	FWD	6	N/A	6	N/A	6	N/A	6	N/A	6	N/A	= RefID * 2 + 1
14	BWD	6	N/A	N/A	6	N/A	6	N/A	6	N/A	6	= RefID * 2 + 2
15	FWD	7	N/A	7	N/A	7	N/A	7	N/A	7	N/A	= RefID * 2 + 1
16	BWD	7	N/A	N/A	7	N/A	7	N/A	7	N/A	7	= RefID * 2 + 2
17	FWD	8	N/A	8	N/A	8	N/A	8	N/A	8	N/A	= RefID * 2 + 1
18	BWD	8	N/A	N/A	8	N/A	8	N/A	8	N/A	8	= RefID * 2 + 2
19	FWD	9	N/A	9	N/A	9	N/A	9	N/A	9	N/A	= RefID * 2 + 1
20	BWD	9	N/A	N/A	9	N/A	9	N/A	9	N/A	9	= RefID * 2 + 2
21	FWD	10	N/A	10	N/A	10	N/A	10	N/A	10	N/A	= RefID * 2 + 1
22	BWD	10	N/A	N/A	10	N/A	10	N/A	10	N/A	10	= RefID * 2 + 2
23	FWD	11	N/A	11	N/A	11	N/A	11	N/A	11	N/A	= RefID * 2 + 1
24	BWD	11	N/A	N/A	11	N/A	11	N/A	11	N/A	11	= RefID * 2 + 2
25	FWD	12	N/A	12	N/A	12	N/A	12	N/A	12	N/A	= RefID * 2 + 1
26	BWD	12	N/A	N/A	12	N/A	12	N/A	12	N/A	12	= RefID * 2 + 2
27	FWD	13	N/A	13	N/A	13	N/A	13	N/A	13	N/A	= RefID * 2 + 1
28	BWD	13	N/A	N/A	13	N/A	13	N/A	13	N/A	13	= RefID * 2 + 2
29	FWD	14	N/A	14	N/A	14	N/A	14	N/A	14	N/A	= RefID * 2 + 1
30	BWD	14	N/A	N/A	14	N/A	14	N/A	14	N/A	14	= RefID * 2 + 2
31	FWD	15	N/A	15	N/A	15	N/A	15	N/A	15	N/A	= RefID * 2 + 1
32	BWD	15	N/A	N/A	15	N/A	15	N/A	15	N/A	15	= RefID * 2 + 2

Interlaced Content

	Driver View			Universal Input M1.6 RefIDs (4b Value per Block)								Conversion
BTI	Direction	Number	Field	FWD 0	BWD 0	FWD 1	BWD 1	FWD 2	BWD 2	FWD 3	BWD 3	BTI Equation
0	Source	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	= From input
1	FWD	0	Top	0	N/A	0	N/A	0	N/A	0	N/A	= RefID * 2 + 1
2	BWD	0	Top	N/A	0	N/A	0	N/A	0	N/A	0	= RefID * 2 + 2
3	FWD	0	Bot	1	N/A	1	N/A	1	N/A	1	N/A	= RefID * 2 + 1
4	BWD	0	Bot	N/A	1	N/A	1	N/A	1	N/A	1	= RefID * 2 + 2
5	FWD	1	Top	2	N/A	2	N/A	2	N/A	2	N/A	= RefID * 2 + 1
6	BWD	1	Top	N/A	2	N/A	2	N/A	2	N/A	2	= RefID * 2 + 2
7	FWD	1	Bot	3	N/A	3	N/A	3	N/A	3	N/A	= RefID * 2 + 1
8	BWD	1	Bot	N/A	3	N/A	3	N/A	3	N/A	3	= RefID * 2 + 2
9	FWD	2	Top	4	N/A	4	N/A	4	N/A	4	N/A	= RefID * 2 + 1
10	BWD	2	Top	N/A	4	N/A	4	N/A	4	N/A	4	= RefID * 2 + 2
11	FWD	2	Bot	5	N/A	5	N/A	5	N/A	5	N/A	= RefID * 2 + 1
12	BWD	2	Bot	N/A	5	N/A	5	N/A	5	N/A	5	= RefID * 2 + 2
13	FWD	3	Top	6	N/A	6	N/A	6	N/A	6	N/A	= RefID * 2 + 1
14	BWD	3	Top	N/A	6	N/A	6	N/A	6	N/A	6	= RefID * 2 + 2
15	FWD	3	Bot	7	N/A	7	N/A	7	N/A	7	N/A	= RefID * 2 + 1
16	BWD	3	Bot	N/A	7	N/A	7	N/A	7	N/A	7	= RefID * 2 + 2
17	FWD	4	Top	8	N/A	8	N/A	8	N/A	8	N/A	= RefID * 2 + 1
18	BWD	4	Top	N/A	8	N/A	8	N/A	8	N/A	8	= RefID * 2 + 2
19	FWD	4	Bot	9	N/A	9	N/A	9	N/A	9	N/A	= RefID * 2 + 1
20	BWD	4	Bot	N/A	9	N/A	9	N/A	9	N/A	9	= RefID * 2 + 2
21	FWD	5	Top	10	N/A	10	N/A	10	N/A	10	N/A	= RefID * 2 + 1
22	BWD	5	Top	N/A	10	N/A	10	N/A	10	N/A	10	= RefID * 2 + 2
23	FWD	5	Bot	11	N/A	11	N/A	11	N/A	11	N/A	= RefID * 2 + 1
24	BWD	5	Bot	N/A	11	N/A	11	N/A	11	N/A	11	= RefID * 2 + 2
25	FWD	6	Top	12	N/A	12	N/A	12	N/A	12	N/A	= RefID * 2 + 1
26	BWD	6	Top	N/A	12	N/A	12	N/A	12	N/A	12	= RefID * 2 + 2
27	FWD	6	Bot	13	N/A	13	N/A	13	N/A	13	N/A	= RefID * 2 + 1
28	BWD	6	Bot	N/A	13	N/A	13	N/A	13	N/A	13	= RefID * 2 + 2
29	FWD	7	Top	14	N/A	14	N/A	14	N/A	14	N/A	= RefID * 2 + 1
30	BWD	7	Top	N/A	14	N/A	14	N/A	14	N/A	14	= RefID * 2 + 2
31	FWD	7	Bot	15	N/A	15	N/A	15	N/A	15	N/A	= RefID * 2 + 1
32	BWD	7	Bot	N/A	15	N/A	15	N/A	15	N/A	15	= RefID * 2 + 2

Glossary of Messages

This section describes the glossary of messages in regard to Media Sampler.

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 = [8, 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, 128] Note: Please make sure the reference windows are not completely outside of the video frame. In 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	Reserved: MBZ
	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.

DWord	Bits	Description
		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.
	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

DWord	Bits	Description
	18	<p>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 considered in FBR (a shape is either FWD or BWD as input of FBR, output however could change to BI if BME is enabled).</p> <p>0 = BME enabled 1 = BME disabled ValidMsgType = FBR</p>
	17	<p>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 marginally so don't enable it unless the transform is necessary.</p> <p>0 = FT disabled 1 = FT enabled ValidMsgType = SIC</p>
	16	<p>Process Inter Chroma Pixels Mode (InterChroaZmaMode): This bit switches the inter operations from luma mode to chroma mode.</p> <p>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.</p> <p>MBMode is always 8x8.</p> <p>MBSUBShape is either 8x8 or 4x4 indicated by LSB[1:0]. Bits[7:2] are MBZ.</p> <p>For MBSUBShape of 4x4, SubPredMode is mapped to each 4x4 shape.</p> <p>Only 8x8 and 4x4 ModeCost are valid.</p> <p>Source block size is ignored.</p> <p>Streamin/streamout distortions are overloaded on 16x16 (Chroma8x8) and 8x8 (Chroma4x4).</p> <p>BilinearEnable is ignored (Chroma can only perform bilinear filtering)</p> <p>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).</p> <p>ValidMsgType = SIC, IME, FBR</p>
	15	<p>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.</p> <p>0 – Frame or field cache lines according to RefAccess 1 – Always frame cache lines</p> <p>ValidMsgType = IME, IDM</p>
	14	<p>Skip Mode Type</p> <p>For B_DIRECT_16x16, both motion vectors of the skip center pair 0 are used.</p> <p>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).</p> <p>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)</p> <p>Note: SkipModeType should be programmed to 1MVP for non-16x16 Source size.</p> <p>ValidMsgType = SIC</p>

DWord	Bits	Description										
	13:12	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										
	11	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										
	10:8	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><tr><td>001b</td><td>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.</td></tr><tr><td>011b</td><td>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.</td></tr><tr><td>111b</td><td>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.</td></tr></table>	Code	Mode	000b	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.	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.
Code	Mode											
000b	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.											
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.											

DWord	Bits	Description
		<p>If BiSubMbPartMask is set to 1111'b, bidirectional search is disabled. VME outputs only the best unidirectional search results. Otherwise, BME is performed.</p> <p>Note that bidirectional search and sub-pel refinement are orthogonal features that can be enabled independently.</p>
	7	<p>Reference Access (RefAccess)</p> <p>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.</p> <p>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.</p> <p>0: Frame based 1: Field based</p> <p>ValidMsgType = SIC, IME, FBR, IDM</p>
	6	<p>Source Access (SrcAccess)</p> <p>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.</p> <p>0: Frame based 1: Field based</p> <p>ValidMsgType = SIC, IME, FBR, IDM</p>
	5:4	<p>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).</p> <p>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</p> <p>ValidMsgType = IME, FBR</p>
	3	Reserved: MBZ
	2	Reserved: MBZ
	1:0	<p>Source Block Size (SrcSize)</p> <p>This field defines how the 16x16 source block is formed. When Source Block Size is less than 16x16, SU larger than 4x4 is used.</p> <p>00b: 16x16 01b: 16x8 10b: Reserved (for 8x16) 11b: 8x8</p> <p>ValidMsgType = SIC, IME, FBR, IDM</p> <p>Note: For IDM message, the source block size should be always programmed to 16x16.</p>
M0.2	31:16	Source Y (SrcY)

DWord	Bits	Description		
		<p>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.</p> <table><tr><th>Restriction</th></tr><tr><td><p>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.</p><p>ValidMsgType = SIC, IME, FBR, IDM</p></td></tr></table> <p>Format = U16</p>	Restriction	<p>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.</p> <p>ValidMsgType = SIC, IME, FBR, IDM</p>
Restriction				
<p>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.</p> <p>ValidMsgType = SIC, IME, FBR, IDM</p>				
	15:0	<p>Source X (SrcX)</p> <p>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.</p> <p>The source block must be within the source picture starting at any integer grid.</p> <p>For SIC messages where Intra Compute Type is set to 00 (Luma + Chroma enabled), SrcX must be a multiple of 2.</p> <table><tr><td><p>ValidMsgType = SIC, IME, FBR, IDM</p></td></tr></table> <p>Format = U16</p>	<p>ValidMsgType = SIC, IME, FBR, IDM</p>	
<p>ValidMsgType = SIC, IME, FBR, IDM</p>				
M0.1	31:16	<p>Reference 1 Y Delta (Ref1Y)</p> <p>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.</p> <table><tr><th>Restriction</th></tr><tr><td><p>The Y address restriction is removed.</p><p>ValidMsgType = IME</p><p>Format = S15</p><p>Hardware Range: [-2048 to 2047]</p></td></tr></table> <p>Format = U16</p>	Restriction	<p>The Y address restriction is removed.</p> <p>ValidMsgType = IME</p> <p>Format = S15</p> <p>Hardware Range: [-2048 to 2047]</p>
Restriction				
<p>The Y address restriction is removed.</p> <p>ValidMsgType = IME</p> <p>Format = S15</p> <p>Hardware Range: [-2048 to 2047]</p>				
	15:0	<p>Reference 1 X Delta (Ref1X)</p> <p>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.</p> <p>The resulting reference region is allowed to be outside the picture. Pixel replication is applied to generate out of bound reference pixels.</p> <p>This field is only valid when dual reference mode is selected.</p> <p>Note: For search control=3, this must equal Ref0X.</p> <p>ValidMsgType = IME</p> <p>Format = S15</p> <p>Hardware Range: [-2048 to 2047]</p>		
M0.0	31:16	<p>Reference 0 Y Delta (Ref0Y)</p> <p>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.</p> <table><tr><th>Restriction</th></tr><tr><td><p>The Y address restriction is removed.</p><p>ValidMsgType = IME, IDM</p><p>Format = S15</p><p>Hardware Range: [-2048 to 2047]</p></td></tr></table> <p>Format = U16</p>	Restriction	<p>The Y address restriction is removed.</p> <p>ValidMsgType = IME, IDM</p> <p>Format = S15</p> <p>Hardware Range: [-2048 to 2047]</p>
Restriction				
<p>The Y address restriction is removed.</p> <p>ValidMsgType = IME, IDM</p> <p>Format = S15</p> <p>Hardware Range: [-2048 to 2047]</p>				

DWord	Bits	Description
	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] 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), 128x16 (shape16x16), and 32x16 (shape16x16) are supported. Search control[2:0] defaults to single ref and single start. Luma only.
	22	RefID Cost Mode Select (RefIDCostMode) Selects the RefID costing mode. 0 = Mode0 (AVC) 1 = Mode1 (linear) <div>ValidMsgType = SIC, IME, FBR, IDM</div>
	21	Enable AC-Only HAAR (AOnlyHAAR) 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) <div> <div>Restriction</div> This bit enables weighted SAD\HAAR. 0: No weighted-SAD </div>

DWord	Bits	Description
		<p>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</p>
	19	<p>Source Field Polarity Select (SrcFieldPolarity) If SrcAccess = 1 (M0.3-6), meaning field based, then 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: $\text{SrcY} * 2 + \text{SrcFieldPolarity}$ Else, this field is ignored by the hardware. ValidMsgType = SIC, IME, FBR, IDM Format = U1</p>
	18	<p>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</p>
	17:16	<p>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 Format = U2</p>

DWord	Bits	Description																
	15:8	<p>Macroblock Intra Structure (MbIntraStruct): This is a bitmask that specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.</p> <p>Note: The user must set Bit6=Bit5.</p> <p>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.</p> <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> <p>ValidMsgType = SIC</p>	Bit	MotionVerticalFieldSelect Index	7	Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)	6	Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)	5	IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)	4	IntraPredAvailFlagB – B (Upper neighbor)	3	IntraPredAvailFlagC – C (Upper left neighbor)	2	IntraPredAvailFlagD – D (Upper right neighbor)	1:0	Reserved: MBZ (ChromaIntraPredMode)
	Bit	MotionVerticalFieldSelect Index																
	7	Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)																
	6	Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)																
	5	IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)																
	4	IntraPredAvailFlagB – B (Upper neighbor)																
	3	IntraPredAvailFlagC – C (Upper left neighbor)																
	2	IntraPredAvailFlagD – D (Upper right neighbor)																
	1:0	Reserved: MBZ (ChromaIntraPredMode)																
	7	<p>Luma Intra Source Corner Swap (IntraCornerSwap): This field specifies the format of the intra luma neighbor pixel format in the message.</p> <p>0: Top neighbors are in sequential order.</p> <p>1: Left-top corner is swapped with the last left-edge neighbor.</p> <p>ValidMsgType = SIC</p>																
6	<p>Non Skip MB Mode Cost Added (NonSkipModeAdded)</p> <p>This field indicates that the distortion of the survived motion vectors becomes non-skip, and the MB mode cost is added to its distortion.</p> <p>ValidMsgType = SIC</p>																	
5	<p>Non Skip Zero MV Cost Added (NonSkipZMvAdded)</p> <p>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.</p> <p>ValidMsgType = SIC</p>																	
4:0	<p>Luma Intra Partition Mask (IntraPartMask)</p> <p>This field specifies which Luma Intra partition is enabled/disabled for intra mode decision.</p> <p>xxx1: luma_intra_16x16 disabled</p> <p>xxx1x: luma_intra_8x8 disabled</p> <p>xx1xx: luma_intra_4x4 disabled</p> <p>Note: For SIC message with IntraComputeType == 00 or 01, at least 1 partition must be enabled.</p> <p>Bits [4:3] MBZ</p> <p>ValidMsgType = SIC</p>																	
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																

DWord	Bits	Description		
	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. <table border="1"><tr><td>Restriction</td></tr><tr><td>Note: All 4 Bwd RefID are ignored by HW for IDM message type. Format = U4 ValidMsgType = SIC, IME, FBR, IDM</td></tr></table>	Restriction	Note: All 4 Bwd RefID are ignored by HW for IDM message type. Format = U4 ValidMsgType = SIC, IME, FBR, IDM
	Restriction			
	Note: All 4 Bwd RefID are ignored by HW for IDM message type. Format = U4 ValidMsgType = SIC, IME, FBR, IDM			
31:16	Reserved: MBZ			
15:0	Reserved: MBZ			
	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)		
	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 4 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 for the first 16x16 weighted-SAD result: 16x16_0 Weighted-SAD Control Mapping:		

DWord	Bits	Description
		0145 2367 89CD ABEF The HW will horizontally, vertically, and diagonally map these weights from M1.3 31:0 to produce the weights for the other 3 16x16 macroblock results. 16x16_1 Weighted-SAD Control Mapping (Horizontal reflection): 5410 7632 DC98 FEBA 16x16_2 Weighted-SAD Control Mapping (Vertical reflection): ABEF 89CD 2367 0145 16x16_3 Weighted-SAD Control Mapping (Diagonal reflection): FEBA DC98 7632 5410 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

DWord	Bits	Description
	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]
	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	Reserved: MBZ
	30	Reserved: MBZ
	29	Ref pixel bias enable If set perform following to reference pixel: »1+64 ValidMsg type - IME, SIC, FBR, IDM
	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
	27:24	Reserved: MBZ
	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

DWord	Bits	Description
		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 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	Reserved: MBZ
	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

DWord	Bits	Description
	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 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
	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]

DWord	Bits	Description
	15:0	SIC Forward Transform Coeff Threshold Matrix[0] Values of the threshold matrix[0..6] are provided here. Matrix[0] contains the DC threshold for the Forward Transform Skip check. It has increased precision vs. the other thresholds due to the larger size of DC coefficients. 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 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

DWord	Bits	Description
	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
	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) <div style="border: 1px solid black; padding: 2px; margin-top: 5px;"> Note ValidMsgType = SIC, IME, FBR, IDM </div>
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

DWord	Bits	Description
M2.1	31:24	Mode 7 Cost MODE_INTER_4x4q 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) 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

DWord	Bits	Description
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 <div>All 4 Bwd Cost Center Deltas are ignored by HW for IDM message type.</div> 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. <div>ValidMsgType = SIC, IME, FBR, IDM</div> 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 <div>ValidMsgType = SIC, IME, FBR, IDM</div> 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:

Value	Any Description
4	BDW added 1 additional universal message phase.

DWord	Bits	Name
WX+0.7	31:0	Ref1 SkipCenter 3 Delta XY <u>Ref1 Skip Center 3 Delta Y:</u> 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] <u>Ref1SkipCenter3 Delta X:</u> 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]

DWord	Bits	Name
WX+0.6	31:0	Ref0 SkipCenter 3 Delta XY (for definition see M3.7)
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 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:
	31:25	Reserved: MBZ

DWord	Bits	Name
WX+1.0	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. 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.

DWord	Bits	Name
	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 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	<p>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.</p> <p>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.</p> <p>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.</p> <p>This set of intra predictor mode for neighbor macroblocks are only used for INTRA8x8 and INTRA4x4 modes.</p> <p>Format : U4 (The value of this field is defined in Definition of Intra4x4PredMode which is the same as that in Definition of Intra8x8PredMode.)</p>
WX+2.3	31:24	<p>Corner Neighbor pixel 1. Its content depends on IntraCornerSwap field. It swaps with Corner Neighbor pixel 0.</p> <p>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</p> <p>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.</p>
	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

DWord	Bits	Name
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:

Value	Any Description
4	BDW added 1 additional universal message phase.

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
	23:20	Rec0 Shape 8x8_1 RefID
	19:16	Rec0 Shape 8x8_0 RefID
	15:12	Rec0 Shape 8x16_1 RefID

DWord	Bits	Name
	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 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)

DWord	Bits	Name
	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
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

DWord	Bits	Name
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
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:

Value	Any Description
4	BDW added 1 additional universal message phase.

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
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

DWord	Bits	Name
WX+2.5	31:0	Ref1 Sub-block XY 10
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:

Value	Any Description
4	BDW added 1 additional universal message phase.

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
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
	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 [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
	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
	24	EarlyIMESop Occurred: This bit is set if the EarlyIMESop threshold is satisfied and IME discontinues searching. Format: U1 ValidMsgType = IME

DWord	Bits	Description																
	23:16	<p>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.</p> <p>This field is derived from sub_mb_type for a BP_8x8 macroblock.</p> <p>This field is derived from MbType for a non-BP_8x8 inter macroblock, and carries redundant information as MbType.</p> <p>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.</p> <p>Bits [1:0]: SubMbPredMode[0] Bits [3:2]: SubMbPredMode[1] Bits [5:4]: SubMbPredMode[2] Bits [7:6]: SubMbPredMode[3] ValidMsgType = SIC, IME, FBR</p>																
	15:8	<p>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.</p> <p>This field is derived from sub_mb_type for a BP_8x8 or equivalent macroblock.</p> <p>This field is forced to 0 for a non-BP_8x8 inter macroblock, and effectively carries redundant information as MbType.</p> <p>This field is only valid if InterMbMode is INTER8x8; otherwise, it is set to zero.</p> <p>Bits [1:0]: SubMbShape[0] Bits [3:2]: SubMbShape[1] Bits [5:4]: SubMbShape[2] Bits [7:6]: SubMbShape[3] ValidMsgType = SIC, IME, FBR</p>																
	7:0	<p>Macroblock Intra Structure (MbIntraStruct): This is a bitmask that specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.</p> <p>This field is simply copied from the input message (to reduce software overhead of forming the output message to PAK).</p> <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> <p>Note: This 8b field is MBZ when IntraComputeType == 1X (when intra is disabled). ValidMsgType = SIC</p>	Bits	MotionVerticalFieldSelect Index	7	Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)	6	Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)	5	IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)	4	IntraPredAvailFlagB – B (Upper neighbor)	3	IntraPredAvailFlagC – C (Upper left neighbor)	2	IntraPredAvailFlagD – D (Upper right neighbor)	1:0	ChromaIntraPredMode
Bits	MotionVerticalFieldSelect Index																	
7	Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff)																	
6	Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff)																	
5	IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)																	
4	IntraPredAvailFlagB – B (Upper neighbor)																	
3	IntraPredAvailFlagC – C (Upper left neighbor)																	
2	IntraPredAvailFlagD – D (Upper right neighbor)																	
1:0	ChromaIntraPredMode																	
W0.5	31:16	LumaIntraPredModes[3]																
	15:0	LumaIntraPredModes[2]																

DWord	Bits	Description
W0.4	31:16	LumaIntraPredModes[1]
	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
	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
	15:0	InterDistortion This field provides the best inter distortion. Format = U16 ValidMsgType = SIC, IME, FBR
W0.1	31:27	Reserved: MBZ
	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
	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

DWord	Bits	Description
	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
	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	31:29	Reserved: MBZ
	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. Format: U5, valid from 0 to 32 ValidMsgType = SIC, IME, FBR
	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
	22:21	Reserved: MBZ
	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
	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

DWord	Bits	Description
	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
	13	Reserved: MBZ
	12:8	InterMbType 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
	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
	6	Reserved: MBZ
	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
	3:2	Reserved: MBZ
	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 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

DWord	Bits	Description
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
	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
	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 1 st weighting pattern (16x16_0) W1.1 from 31:0 = Reserved W1.2 from 31:0 = Minimum 16x16 distortion when applying 2 nd weighting pattern (16x16_1) W1.3 from 31:0 = Reserved W1.4 from 31:0 = Minimum 16x16 distortion when applying 3 rd weighting pattern (16x16_2) W1.5 from 31:0 = Reserved W1.6 from 31:0 = Minimum 16x16 distortion when applying 4 th weighting pattern (16x16_3) W1.7 from 31:0 = Reserved W2 through W4 = 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

DWord	Bits	Description
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. 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
	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 1 st weighting pattern (16x16_0 with corresponding MV at W1.0) W5.0 from 31:16 = Minimum 16x16 distortion when applying 2 nd weighting pattern (16x16_1 with corresponding MV at W1.2) W5.1 from 15:0 = Minimum 16x16 distortion when applying 3 rd weighting pattern (16x16_2 with corresponding MV at W1.4) W5.1 from 31:16 = Minimum 16x16 distortion when applying 4 th weighting pattern (16x16_3 with corresponding MV at W1.6) W5.7 to W5.2 = Reserved

DWord	Bits	Description
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
	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
	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
	15:0	Block 0 Chroma Cb 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
	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
	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
	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
	15:0	Block 2 Luma Coeff Magnitude Clip Sum

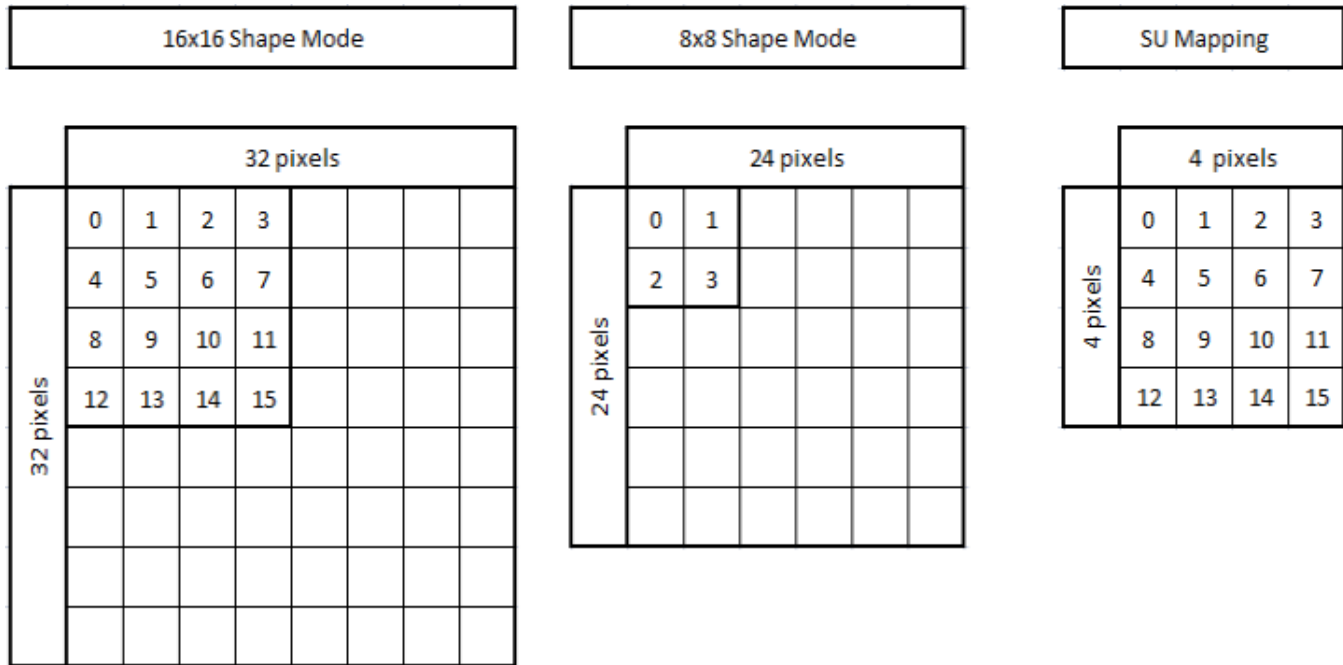
DWord	Bits	Description
W6.2	31:16	Block 1 Luma Coeff Magnitude Clip Sum
	15:0	Block 0 Luma Coeff Magnitude Clip Sum Sum of how much all the coefficients across 1 block exceeded their respective threshold. Note: This field is MBZ when SkipModeEn is not set (when skip is disabled). Format = U16 ValidMsgType = SIC
W6.1	31:24	Block 3 Luma NZC
	23:16	Block 2 Luma NZC
	15:8	Block 1 Luma NZC
	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
	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 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
	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. 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
	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

DWord	Bits	Description
	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
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
	31:16	Distortion Mesh Block 0 SU0 Search Point 5

DWord	Bits	Description
W0.2	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 1 st 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 2 nd 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

SAMPLER_STATE for Sample_8x8 Message

SAMPLER_STATE has different formats, depending on the message type used. For BDW, the sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

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 AVS Scaling : State is 154DWs
- 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), (upsized 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 footprint 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 footprint 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_8x8_AVS

SAMPLER_STATE_8x8_AVS_COEFFICIENTS

SAMPLER_STATE_8x8_CONVOLVE

SAMPLER_STATE_8x8_ERODE_DILATE_MINMAXFILTER

Media Object Dispatch Pseudocode

```
// Variables:
Frame Height in pixels => frame_height
Frame Width in pixels => frame_width
Frame Height in Blocks => fh
Frame Width in Blocks => fw
Block Height in Pixels => block_height = Interlaced ? 4 : 8

// Code:
fw = frame_width / 16;
fh = frame_height / block_height;
```

Calculate Residual Blocks Pseudocode

```
If ( fh % (2**stride) ) != 0 {
    Y_Blocks_Remainder = (fh % (2**stride))
    If ( Y_Blocks_Remainder > (2**stride) / 2 ) {
        Y_Blocks_Remainder_HS1 = (2**stride) / 2
        Y_Blocks_Remainder_HS2 = Y_Blocks_Remainder - (2**stride) / 2
    }
    Else {
        Y_Blocks_Remainder_HS1 = Y_Blocks_Remainder
        Y_Blocks_Remainder_HS2 = 0
    }
}
Else {
    Y_Blocks_Remainder_HS1 = 0
```

```
Y_Blocks_Remainder_HS2 = 0  
}
```

Dispatch Media Object Pseudocode

```
total_media_obj_cnt = fw * fh;
remainder_media_obj_cnt_HS1 = fw * Y_Blocks_Remainder_HS1;
remainder_media_obj_cnt_HS2 = fw * Y_Blocks_Remainder_HS2;

ping_pong_media_obj_cnt = total_media_obj_cnt - (remainder_media_obj_cnt_HS1 +
remainder_media_obj_cnt_HS1);

for ( i = 0; i < ping_pong_media_obj_cnt; i++ ) {
    if ( i % 2 == 0 ) {
        dispatch_media_object_hs1;
    }
    else {
        dispatch_media_object_hs2;
    }
}

for ( i = 0; i < remainder_media_obj_cnt_HS1; i++ ) {
    dispatch_media_object_hs1;
}

for ( i = 0; i < remainder_media_obj_cnt_HS2; i++ ) {
    dispatch_media_object_hs2;
}
```

SIMD32/64 Messages

Initiating Message

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

DWord	Bits	Description																				
M1.7	31:28	<div>Functionality</div> <table><tr><th>Value</th><th>Function</th></tr><tr><td>0000b</td><td>AVS scaling</td></tr><tr><td>0001b</td><td>Convolve</td></tr><tr><td>0010b</td><td>MINMAX</td></tr><tr><td>0011b</td><td>MINMAXFILTER</td></tr><tr><td>0100b</td><td>ERODE</td></tr><tr><td>0101b</td><td>Dilate</td></tr><tr><td>0110b</td><td>BoolCentroid / BoolSum</td></tr><tr><td>0111b</td><td>Centroid</td></tr><tr><td>Other</td><td>Reserved</td></tr></table>	Value	Function	0000b	AVS scaling	0001b	Convolve	0010b	MINMAX	0011b	MINMAXFILTER	0100b	ERODE	0101b	Dilate	0110b	BoolCentroid / BoolSum	0111b	Centroid	Other	Reserved
Value	Function																					
0000b	AVS scaling																					
0001b	Convolve																					
0010b	MINMAX																					
0011b	MINMAXFILTER																					
0100b	ERODE																					
0101b	Dilate																					
0110b	BoolCentroid / BoolSum																					
0111b	Centroid																					
Other	Reserved																					
	27	<div>IEF Bypass</div> <div>Ignored for all functions as it will always be IEF bypass except for AVS scaling operation.</div>																				
	26:23	<div>Control</div> <div>[26:25] Message_Seq:</div> <div>When Functionality is AVS:</div> <div>11: 4x4 (only supported with Shuffle_OutputWriteback=1)</div> <div>10: 16x8</div> <div>01: 8x4 (only supported with Shuffle_OutputWriteback=1)</div> <div>00: 16x4</div> <div>When Functionality is Convolve/MinMaxFilter/LBP Correlation/LBP Creation:</div> <div>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.</div> <div>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.</div> <div>0x: 16x4 – except for 1-D Horizontal this will be 4x16; illegal for 1 pixel convolve.</div> <div>When Functionality is Erode/Dilate:</div> <div>11: 32x1 – This would be used in case we require the operation in scattered regions and don’t want to waste the throughput.</div> <div>10: 64x1 – This would be used in case we require the operation in scattered regions and don’t want to waste the throughput.</div> <div>01: 32x4</div> <div>00: 64x4</div> <div>When Functionality is MINMAX/BoolCentroid/Centroid</div> <div>1x: These modes should always force this to 1 since we don’t have SI sequencing.</div> <div>0x: Unexpected behavior</div> <div>%2</div>																				

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	
	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.

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. Parameter 0 is required except for the sampleinfo message for , which has no parameter 0.

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	Project
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.

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	dvdxd	dvdxd	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	dvdxd	dvdxd	drdx	drdy	ref
11100	ld2dms_w	u	v	r	lod	si	mcs	mcs				
11101	ld_mcs	u	v	r	lod							
11110	ld2dms	u	v	r	lod	si	mcs					

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

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.htm](#).

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	31:0	Ignored
M0.4	31:0	Reserved
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 = DynamicStateOffset[31:5]
	4:1	Ignored
	0	Ignored
M0.2	31:24	Ignored
	23:22	Reserved

DWord	Bits	Description												
	21	Ignored												
	20	Ignored												
	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 <table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Message Header</td></tr><tr><td colspan="2">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.</td></tr></table>	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														
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.														
	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).												
	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. <table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Message Header</td></tr><tr><td colspan="2"><ul style="list-style-type: none">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.</td></tr></table> <table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Message Header</td></tr><tr><td colspan="2">For Sample_8x8 messages, Alpha/Blue/Red channels should be always masked (set to 1) and only Green channel is enabled (set to 0).</td></tr></table>	Programming Note		Context:	Message Header	<ul style="list-style-type: none">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).	
Programming Note														
Context:	Message Header													
<ul style="list-style-type: none">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).														
	14	Blue Write Channel Mask: See Alpha Write Channel Mask.												

DWord	Bits	Description						
	13	Green Write Channel Mask: See Alpha Write Channel Mask.						
	12	Red Write Channel Mask: See Alpha Write Channel Mask.						
	11:8	<p>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.</p> <table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Message Header</td></tr><tr><td colspan="2"><ul style="list-style-type: none">• 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>	Programming Note		Context:	Message Header	<ul style="list-style-type: none">• 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.	
Programming Note								
Context:	Message Header							
<ul style="list-style-type: none">• 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.								
	7:4	<p>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.</p> <table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Message Header</td></tr><tr><td colspan="2"><ul style="list-style-type: none">• 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>	Programming Note		Context:	Message Header	<ul style="list-style-type: none">• 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.	
Programming Note								
Context:	Message Header							
<ul style="list-style-type: none">• 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.								
	3:0	<p>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.</p> <p>This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.</p>						
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
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
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
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

DWord	Bit	Description
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
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
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
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.

DWord	Bit	Description																																
Wn.7:1		Reserved (not written)																																
Wn.0	31:0	<p>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.</p> <p>The bits in this mask correspond to the pixels as follows and they are listed from upper left (MSB) lower right LSB:</p> <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>	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
31	30	29	28	27	26	25	24																											
23	22	21	20	19	18	17	16																											
15	14	13	12	11	10	9	8																											
7	6	5	4	3	2	1	0																											

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
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
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

DWord	Bits	Description
W6.5		Pixel 22 & 20 Blue
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
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
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 35:32 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
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
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

DWord	Bits	Description
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 Convolve and 1 Pixel Convolution and 1D Vertical Convolve

DWord	Bits	Description
W0.7		Pixel 15 & 14 Row 0
W0.6		Pixel 13 & 12 Row 0
W0.5		Pixel 11 & 10 Row 0
W0.4		Pixel 9 & 8 Row 0
W0.3		Pixel 7 & 6 Row 0
W0.2		Pixel 5 & 4 Row 0
W0.1		Pixel 3 & 2 Row 0
W0.0	31:16	Pixel 1 Row 0 Format = 16-bit SINT
	15:0	Pixel 0 Row 0
W1		Pixel [15:0] Row 1 Only for 16x4 Message. Not present for 16x1 and 1x1 message.
W2		Pixel [15:0] Row 2 Only for 16x4 Message. Not present for 16x1 and 1x1 message.
W3		Pixel [15:0] Row 3 Only for 16x4 Message. Not present for 16x1 and 1x1 message.

Sampler_8x8 – Writeback Message for MinMaxFilter

"16 Bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	Pixel 7 Max MSB Row 0 if input is 16bits Pixel 7 Max Row 0 if input is 8bits Reserved if 1x1 message
	23:16	Pixel 7 Max LSB Row 0 if input is 16bits Reserved if 1x1 message (or) 8bit input
	15:8	Pixel 7 Min MSB Row 0 if input is 16bits Pixel 7 Min Row 0 if input is 8bits Reserved if 1x1 message
	7:0	Pixel 7 Min LSB Row 0 if input is 16bits Reserved if 1x1 message (or) 8bit input
W0.6		Pixel 6 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.5		Pixel 5 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.4		Pixel 4 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.3		Pixel 3 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.2		Pixel 2 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.1		Pixel 1 Min/Max Row 0 (Format same as above depending on 8bit or 16bit) Reserved if 1x1 message
W0.0		Pixel 0 Row 0 if input is 16bits [31:0] Pixel 0 Max [15:0] Pixel 0 Min Pixel 0 Row 0 if input is 8bits [31:24] Pixel 0 Max [23:16] Reserved [15:8] Pixel 0 Min [7:0] Reserved

DWord	Bit	Description
		The Min or Max could be disabled. Then the corresponding values is invalid and should be ignored.
W1		Pixel [15:8] Row 0
W2		Pixel [7:0] Row 1 Only for 16x4 Message. Not present for 16x1 message
W3		Pixel [15:8] Row 1 Only for 16x4 Message. Not present for 16x1 message
W4		Pixel [7:0] Row 2 Only for 16x4 Message. Not present for 16x1 message
W5		Pixel [15:8] Row 2 Only for 16x4 Message. Not present for 16x1 message
W6		Pixel [7:0] Row 3 Only for 16x4 Message. Not present for 16x1 message
W7		Pixel [15:8] Row 3 Only for 16x4 Message. Not present for 16x1 message

"8 Bit Full" Output Format Control Mode:

DWord	Bit	Description
W0.7	31:24	Pixel 15 Max Row 0 Reserved if 1x1 message
	23:16	Pixel 14 Max Row 0 Reserved if 1x1 message
	15:8	Pixel 15 Min Row 0 Reserved if 1x1 message
	7:0	Pixel 14 Min Row 0 Reserved if 1x1 message
W0.6		Pixel [13:12] Min/Max Row 0 Reserved if 1x1 message
W0.5		Pixel [11:10] Min/Max Row 0 Reserved if 1x1 message
W0.4		Pixel [9:8] Min/Max Row 0 Reserved if 1x1 message
W0.3		Pixel [7:6] Min/Max Row 0 Reserved if 1x1 message
W0.2		Pixel [5:4] Min/Max Row 0 Reserved if 1x1 message
W0.1		Pixel [3:2] Min/Max Row 0 Reserved if 1x1 message
W0.0		Pixel [1:0] Min/Max Row 0 Reserved if 1x1 message
W1		Pixel [15:0] Row 1 Only for 16x4 Message. Not present for 16x1 message
W2		Pixel [15:0] Row 2 Only for 16x4 Message. Not present for 16x1 message
W3		Pixel [15:0] Row 3 Only for 16x4 Message. Not present for 16x1 message

Sampler_8x8 – Writeback Message for MinMax

DWord	Bit	Description
W0.7	31:0	Reserved
W0.6	31:0	Reserved
W0.5	31:0	Reserved
W0.4	31:0	Reserved
W0.3	31:0	Reserved
W0.2	31:0	Reserved
W0.1	31:0	Reserved
W0.0	31:24	Pixel 0 Max MSB if input is 16bits Pixel 0 Max Row 0 if input is 8bits
	23:16	Pixel 0 Max LSB if input is 16bits Reserved if input is 8bits
	15:8	Pixel 0 Min MSB if input is 16bits Pixel 0 Min Row 0 if input is 8bits
	7:0	Pixel 0 Min LSB if input is 16bits Reserved if input is 8bits

Sampler_8x8 – Writeback Message for Dilate or Erode

DWord	Bit	Description
W0.7	31:0	Pixel [63:32] Row 3 Only for 64x4 message. Not present for 32x4 or 64x1 or 32x1 message
W0.6		Pixel [31:0] Row 3 Only for 64x4 or 32x4 Message. Not present for 64x1 or 32x1 message
W0.5		Pixel [63:32] Row 2 Only for 64x4 message. Not present for 32x4 or 64x1 or 32x1 message
W0.4		Pixel [31:0] Row 2 Only for 64x4 or 32x4 Message. Not present for 64x1 or 32x1 message
W0.3		Pixel [63:32] Row 1 Only for 64x4 message. Not present for 32x4 or 64x1 or 32x1 message
W0.2		Pixel [31:0] Row 1 Only for 64x4 or 32x4 Message. Not present for 64x1 or 32x1 message
W0.1		Pixel [63:32] Row 0 Only for 64x4 or 64x1 Message. Not present for 32x4 or 32x1 message
W0.0		Pixel [31:0] Row 0

Sampler_8x8 – Writeback Message for Centroid

DWord	Bit	Description
W0.7	31:0	Divisor/Sum Column 3
W0.6		jSum Column 3
W0.5		Divisor/Sum Column 2
W0.4		jSum Column 2
W0.3		Divisor/Sum Column 1
W0.2		jSum Column 1
W0.1		Divisor/Sum Column 0
W0.0		jSum Column 0
W1		Divisor/Sum & jSum for Column [7:4]
W2		Divisor/Sum & jSum for Column [11:8]
W3		Divisor/Sum & jSum for Column [15:12]

Sampler_8x8 – Writeback Message for BoolCentroid/BoolSum

DWord	Bit	Description
W0.7	31:0	Reserved
W0.6	31:0	Sum 16b Column 1
W0.5	31:0	jSum 16b Column 1
W0.4	31:0	iSum 16b Column 1
W0.3	31:0	Reserved
W0.2	31:0	Sum 16b Column 0
W0.1	31:0	jSum 16b Column 0
W0.0	31:0	iSum 16b Column 0
W1		Column[3:2] same as above

Sampler_8x8 – Writeback Message for AVS

The writeback message for sample_8x8 consists of up to 16 destination registers for 16x4 and 16x8 message type. 16x8 AVS messages have restrictions mentioned in section SIMD32/64 payload for which limits the writeback to only 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	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
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 47:32 Red
W3		Pixels 63:48 Red
W4		Pixels 15:0 Green
W5		Pixels 31:16 Green
W6		Pixels 47:32 Green
W7		Pixels 63:48 Green
W8		Pixels 15:0 Blue
W9		Pixels 31:16 Blue
W10		Pixels 47:32 Blue
W11		Pixels 63:48 Blue
W12		Pixels 15:0 Alpha
W13		Pixels 31:16 Alpha
W14		Pixels 47:32 Alpha
W15		Pixels 63:48 Alpha
		Below is valid only for 16x8 messages
W16		Pixels 79:64 Red

DWord	Bit	Description
W17		Pixels 95:80 Red
W18		Pixels 111:96 Red
W19		Pixels 127:112 Red
W20		Pixels 79:64 Green
W21		Pixels 95:80 Green
W22		Pixels 111:96 Green
W23		Pixels 127:112 Green
W24		Pixels 79:64 Blue
W25		Pixels 95:80 Blue
W26		Pixels 111:96 Blue
W27		Pixels 127:112 Blue
W28		Pixels 111:96 Alpha
W29		Pixels 127:112 Alpha
W30		Pixels 111:96 Alpha
W31		Pixels 127:112 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
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

DWord	Bits	Description
W6.5		Pixel 22 & 20 Blue
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
W8.0		Pixel 1 & 0 Alpha
W9		Pixel 31:16 Alpha
W10		Pixel 47:32 Alpha
W11		Pixel 63:48 Alpha
		Below DWs is only valid for 16x8 message
W12		Pixel [94,92,90,88,86,84,82,80,78,76,74,72,70,68,66,64] Red
W13		Pixel [126,124,122,120,118,116,114,112,110,108,106,104,102,100,98,96] Red
W14		Pixel [79:64] Green
W15		Pixel [95:80] Green
W16		Pixel [111:96] Green
W17		Pixel [127:112] Green
W18		Pixel [94,92,90,88,86,84,82,80,78,76,74,72,70,68,66,64] Blue
W19		Pixel [126,124,122,120,118,116,114,112,110,108,106,104,102,100,98,96] Blue
W20		Pixel [79:64] Alpha

DWord	Bits	Description
W21		Pixel [95:80] Alpha
W22		Pixel [111:96] Alpha
W23		Pixel [127:112] 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
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 35:32 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
		Below DWs is only valid for 16x8 message
W8		Pixel [95:64] Red
W9		Pixel [127:96] Red
W10		Pixel [95:64] Green
W11		Pixel [127:96] Green
W12		Pixel [95:64] Blue
W13		Pixel [127:96] Blue
W14		Pixel [95:64] Alpha
W15		Pixel [127:96] 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
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
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

DWord	Bits	Description
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
		Below DWs is only valid for 16x8 message
W6.7: 6.4		Pixel [126,124,122,120,118,116,114,112,110,108,106,104,102,100,98,96] Red
W6.3: 6.0		Pixel [94,92,90,88,86,84,82,80,78,76,74,72,70,68,66,64] Red
W7		Pixel [95:64] Green
W8		Pixel [127:96] Green
W9.7: 9.4		Pixel [126,124,122,120,118,116,114,112,110,108,106,104,102,100,98,96] Blue
W9.3 :9.0		Pixel [94,92,90,88,86,84,82,80,78,76,74,72,70,68,66,64] Blue
W10		Pixel [95:64] Alpha
W17		Pixel [127:96] 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.

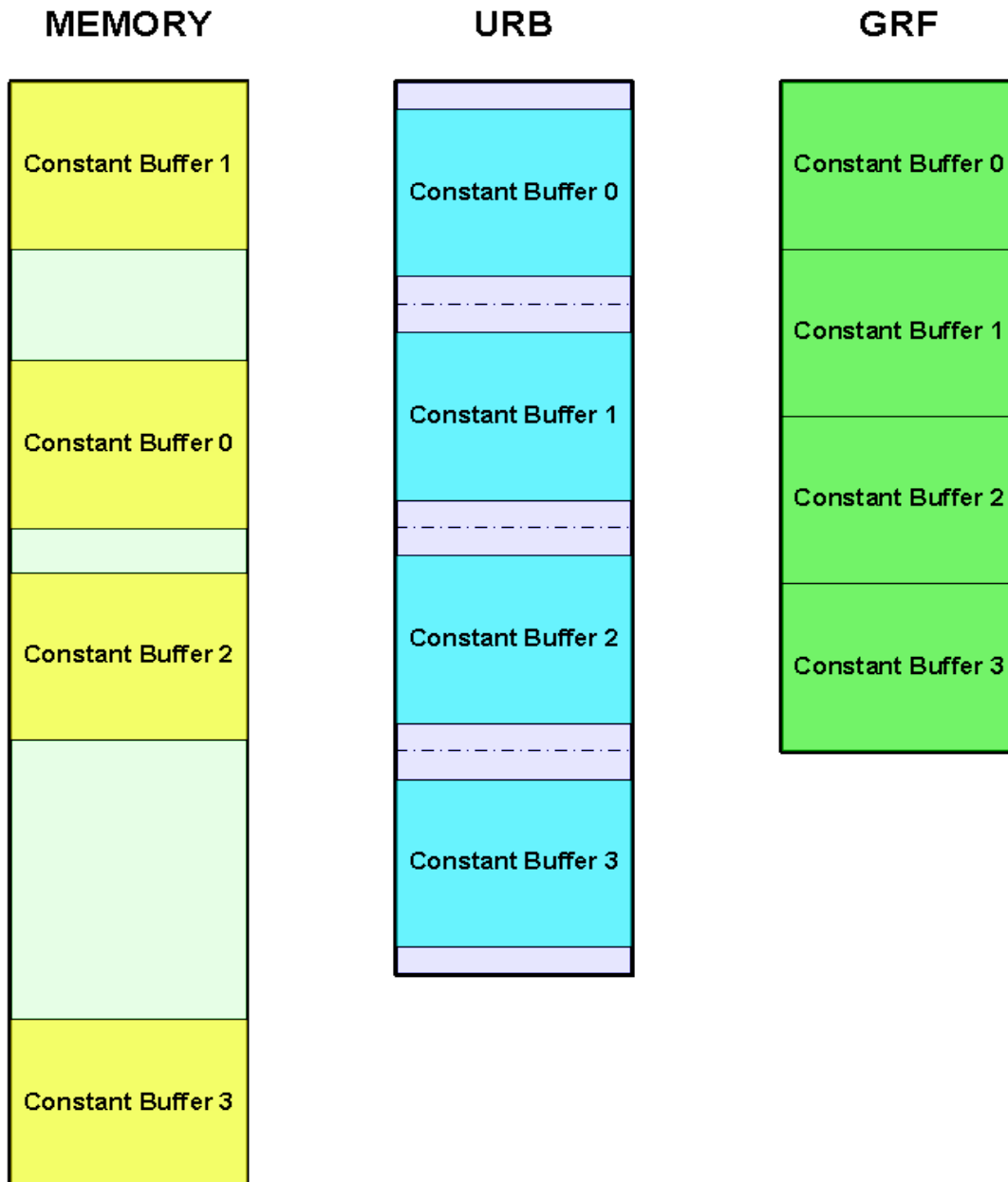
Pipeline Stage	Functions Performed
Command Stream (CS)	The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. Note that the CS stage is shared between the 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
<p>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.</p> <p>Refer to the various 3DSTATE_PUSH_CONSTANT_ALLOC_xx state commands for details regarding the maximum size of the Push Constant and other state programming information.</p>	

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



The bubbles in the URB are caused by the constant buffer in memory starting on a half cacheline and being an even number in length. If the constant buffer starts on an odd cacheline and has an odd number length then there will only be a bubble at the beginning of the buffer in the URB. If the constant buffer in memory starts on a cache line boundary and has an odd number length then the bubble will only be at the end of the constant buffer in the URB. Once the constant buffer is written to the GRF space then all the bubbles will be removed. Software must guarantee that there is enough space in the push constant buffer in the URB to hold one constant buffer from memory. This includes any buffering to write the 512b aligned requests from memory into the URB. Because the L3\$ only supports writes from memory in 512b chunks, the URB may have some bubbles between each constant buffer fetch.

Statistics

Statistics Gathering

The table below describes how the device supports the required API statistics counters.

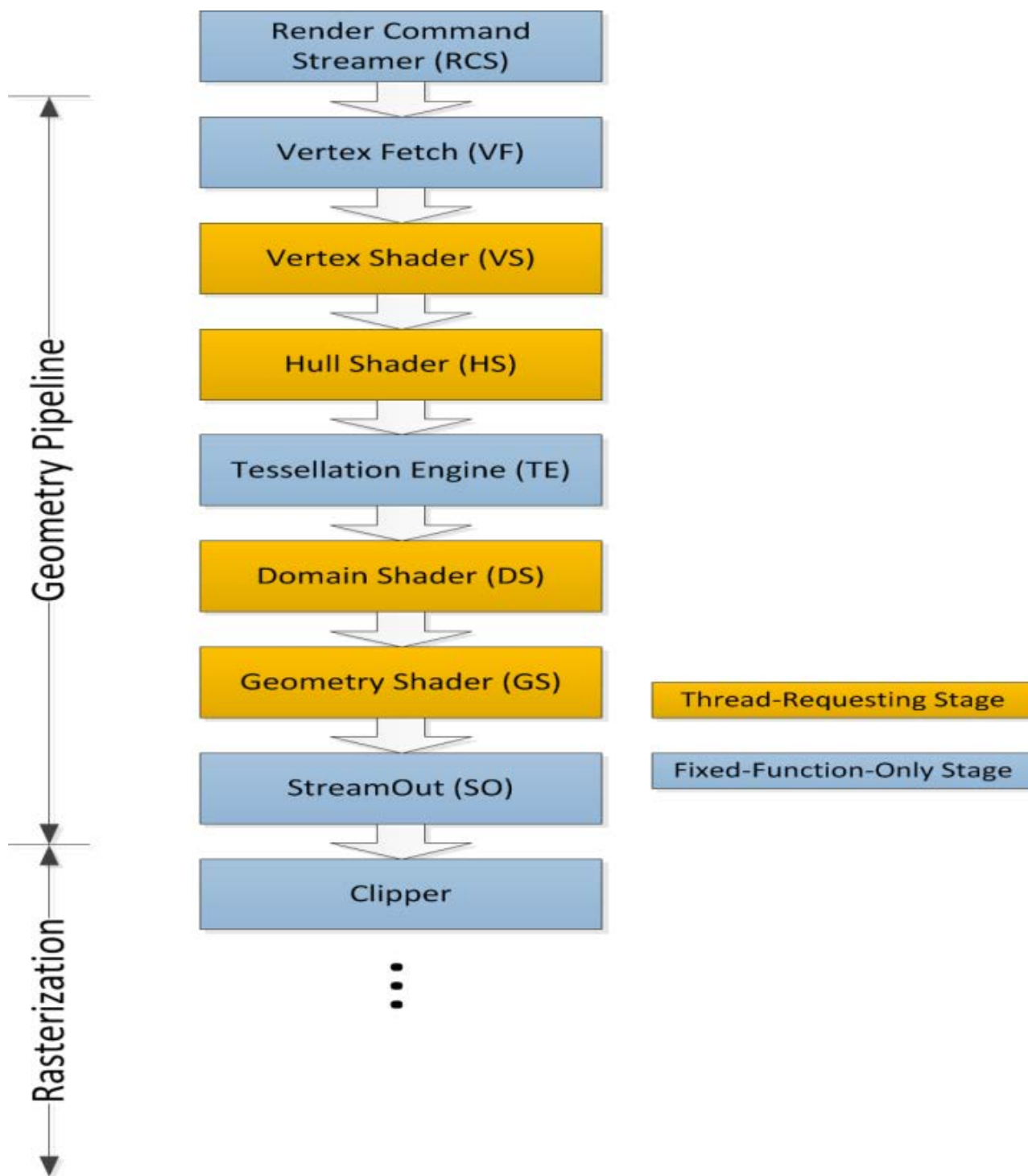
API-level Statistic	HW Support
IAVertices = # of vertices IA generated. May or may not include (a) vertices in partial primitives, (b) unused adjacent-only vertices. Not affected by vertex caching.	VF maintains IA_VERTICES_COUNT . Will include unused adjacent-only vertices. Will not include vertices in partial primitives.
IAPrimitives = # of primitives (objects) IA generated. May or may not include partial primitives.	VF maintains IA_PRIMITIVES_COUNT . Will not include partial primitives. Will not count patch topologies that do not match what the HS or GS expects as input, if enabled (i.e., mismatching patch topologies are discarded by VF).
VSInvocations = # of times VS is executed. May be affected by vertex caching. May or may not include (a) shared vertices in non-indexed strips, (b) vertices in partial primitives, (c) unused adjacent-only vertices.	VS maintains VS_INVOCATION_COUNT . Impacted by vertex caching. Will not include vertices in partial primitives. <u>Will</u> include unused adjacent-only vertices. Will not include shared vertices in non-indexed strips, unless pre-empted. Increments even if VS Function Enable is DISABLED.
HSInvocations = # of patches executed by HS.	HS maintains HS_INVOCATION_COUNT . This gets incremented by 1 for each patch whenever HS is enabled.
DSInvocations = # of times DS is executed to shade a domain point. Allows HW to shade identical domain points multiple times, with the exception of point outputs where only unique domain points can be generated.	DS maintains DS_INVOCATION_COUNT . This is incremented for each domain point passed to a DS thread.
GSInvocations = # of times GS is executed. Obviously does not include partial primitives. May be incremented when StreamOut enabled, even if NULL_GS.	GS maintains GS_INVOCATION_COUNT , incrementing it by GSInvocations Increment Value for each dispatched instance. Will not be incremented if NULL_GS.
GSPrimitives = # of primitives GS generated. Does not include primitives passing through a disabled GS stage. May or may not include partial primitives output by GS.	GS maintains GS_PRIMITIVE_COUNT . GS unit will increment this as it parses the GS thread output. Will <u>not</u> include partial primitives output by GS threads.
NumPrimitivesWritten[<stream#>] = # of complete primitives written to the stream's SO buffer, subject to buffer overflow.	SOL maintains SO_NUM_PRIMS_WRITTEN[0-3] .
PrimitiveStorageNeeded[<stream#>] = # of complete primitives which would have been written to the stream's SO buffer ignoring any overflow.	SOL maintains SO_PRIM_STORAGE_NEEDED[0-3] .

API-level Statistic	HW Support
CInvocations = # of primitives <u>entering</u> rasterization (which starts with the clipper) and isn't affected by any actual clipping. Does not increment when rasterization is disabled (e.g., when StreamOut is the last enabled stage). May or may not include partial primitives.	CL OSB maintains CL_INVOCATION_COUNT . Will not include partial primitives. Note that the SOL (regardless of SO enabled) will discard primitives if rendering is disabled, so these primitives will not reach the CL unit.
CPrimitives = # of primitives output from clipper. I.e., doesn't increment if TrivReject or dropped due to NaNs, increments by 1 if TrivAccept, or increments by number of primitives generated if MustClip. Does not increment when rasterization is disabled. May or may not include partial primitives. Accommodates infinite or no guardband.	SF OSB maintains CL_PRIMITIVES_COUNT . Will not include partial primitives.
PSInvocations = # of times PS is executed, including unlit "helper pixels" within a subspan that need to go through the PS shader to provide 2x2 gradients. Accommodates early depth/stencil. Does not increment if NULL PS. Multisampling: counts pixels shaded if PERPIXEL or samples shaded if PERSAMPLE.	WIZ maintains PS_INVOCATION_COUNT .
Occlusion = # of "visible" multisamples which passed both depth and stencil testing. Doesn't include PS-discarded pixels or oMask/AlphaToCoverage-killed samples. Both (a) a disabled test (depth or stencil) and (b) no bound RT or Depth/Stencil buffer conditions count as always passing.	WIZ & PBE maintain PS_DEPTH_COUNT .

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.



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	
Project:	All
Context:	3D Primitives Overview
<ul style="list-style-type: none"> There are several variants of the basic topologies. These have been introduced to allow slight variations in behavior without requiring a state change. Number of vertices and Dangling Vertices: Topologies have an "expected" number of vertices in order to form complete objects within the topologies (e.g., LINELIST is expected to have an even number of vertices). The actual number of vertices specified in the 3DPRIMITIVE command, and as output from the GS unit, is allowed to deviate from this expected number, in which case any "dangling" vertices are discarded. The removal of dangling vertices is initially performed in the VF unit. To filter out dangling vertices emitted by GS threads, the CLIP unit also performs dangling-vertex removal at its input. 	

3D Primitive Topology Types

3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	<ul style="list-style-type: none"> A list of independent line objects (2 vertices per line). Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELIST_ADJ	<ul style="list-style-type: none"> A list of independent line objects with adjacency information (4 vertices per line). Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. Not valid as output from GS thread.

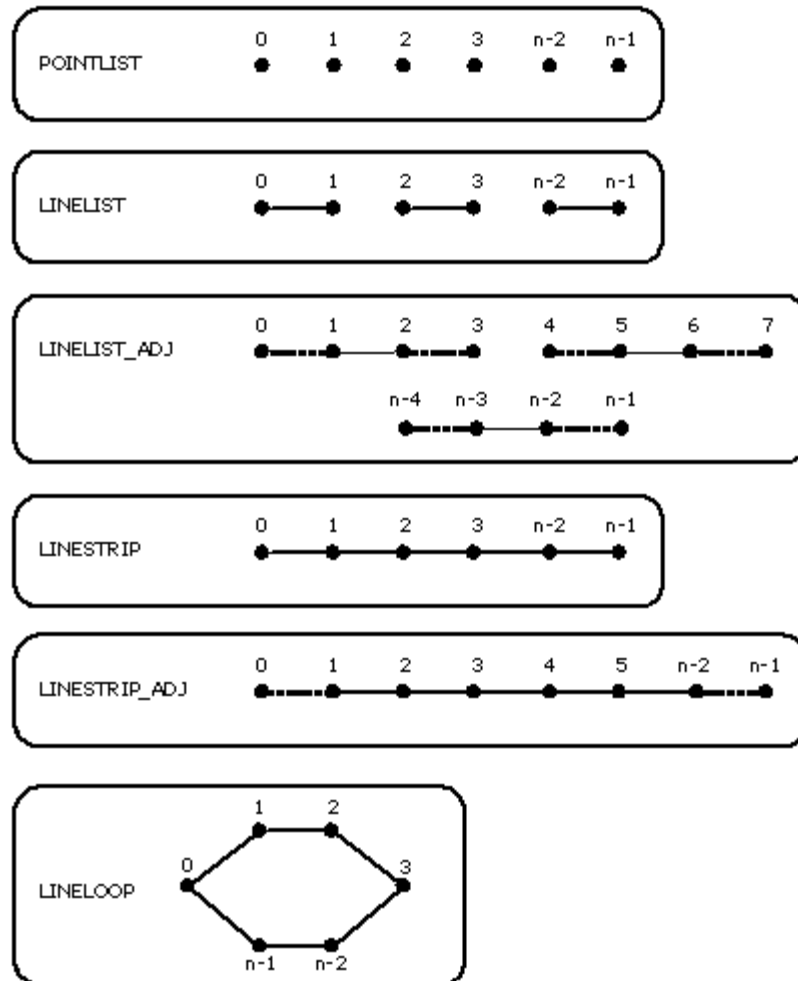
3D Primitive Topology Type (ordered alphabetically)	Description
LINELOOP	<ul style="list-style-type: none"> Similar to a 3DPRIM_LINESTRIP, though the last vertex is connected back to the initial vertex via a line object. The LINELOOP topology is converted to LINESTRIP topology at the beginning of the 3D pipeline. Normal usage expects at least 2 vertices, though incomplete objects are silently ignored. (The 2-vertex case is required by OGL). Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
LINESTRIP	<ul style="list-style-type: none"> A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define a connected line object. Normal usage expects at least 2 vertices, though incomplete objects are silently ignored.
LINESTRIP_ADJ	<ul style="list-style-type: none"> A list of vertices connected such that, after the first vertex, each additional 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Similar to LINESTRIP, except LineStipple (if enabled) is continued (vs. reset) at the start of the primitive topology. This can be used to support line stipple when the API-provided primitive is split across multiple topologies.
LINESTRIP_CONT_BF	Combination of LINESTRIP_BF and LINESTRIP_CONT variations.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	<ul style="list-style-type: none"> Similar to POINTLIST, except treated as “backfacing” during rasterization (stencil test). This can be used to support “point” polygon fill mode when two-sided stencil is enabled.

3D Primitive Topology Type (ordered alphabetically)	Description
POLYGON	<ul style="list-style-type: none"> • Similar to TRIFAN, though the first vertex always provides the “flat-shaded” values (vs. this being programmable through state). • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
QUADLIST	<ul style="list-style-type: none"> • A list of independent quad objects (4 vertices per quad). • The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline. • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored.
QUADSTRIP	<ul style="list-style-type: none"> • A list of vertices connected such that, after the first two vertices, each additional pair of vertices are associated with the previous two vertices to define a connected quad object. • Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored.
RECTLIST	<ul style="list-style-type: none"> • A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle. $V0=LowerRight$, $V1=LowerLeft$, $V2=UpperLeft$. Implied $V3 = V0 - V1 + V2$. • 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).

3D Primitive Topology Type (ordered alphabetically)	Description
TRIFAN	<ul style="list-style-type: none"> Triangle objects arranged in a fan (or polygon). The initial vertex is maintained as a common vertex. After the second vertex, each additional vertex is associated with the previous vertex and the common vertex to define a connected triangle object. Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRIFAN_NOSTIPPLE	<ul style="list-style-type: none"> Similar to TRIFAN, but polygon stipple is not applied (even if enabled). This can be used to support “point” polygon fill mode, under the combination of the following conditions: <ul style="list-style-type: none"> (a) when the frontfacing and backfacing polygon fill modes are different (so the final fill mode is not known to the driver), (b) one of the fill modes is “point” and the other is “solid”, (c) point mode is being emulated by converting the point into a trifan, (d) polygon stipple is enabled. In this case, polygon stipple should not be applied to the points-emulated-as-trifans.
TRILIST	<ul style="list-style-type: none"> A list of independent triangle objects (3 vertices per triangle). Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.
TRILIST_ADJ	<ul style="list-style-type: none"> A list of independent triangle objects with adjacency information (6 vertices per triangle). Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. Not valid as output from GS thread.
TRISTRIP	<ul style="list-style-type: none"> A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.

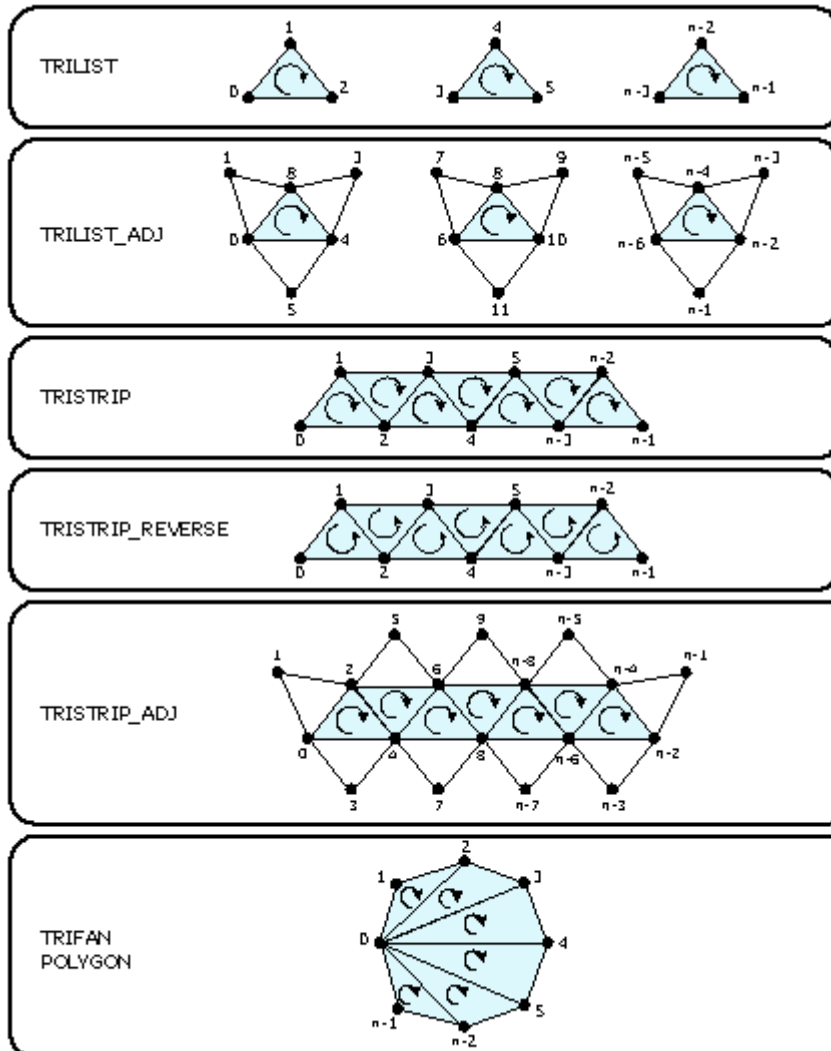
3D Primitive Topology Type (ordered alphabetically)	Description
TRISTRIP_ADJ	<ul style="list-style-type: none"> A list of vertices where the even-numbered (including 0th) vertices are connected such that, after the first two vertex pairs, each additional even-numbered vertex is associated with the last two even-numbered vertices to define a connected triangle object. The odd-numbered vertices are adjacent-only vertices. <p>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.</p> <ul style="list-style-type: none"> Normal usage expects at least 6 vertices, though incomplete objects are silently ignored. Not valid as output from GS thread.
TRISTRIP_REVERSE	<p>Similar to TRISTRIP, though the sense of orientation (winding order) is reversed – this allows SW to break long tristrrips into smaller pieces and still maintain correct face orientations.</p>
BDW: PATCHLIST_n	<p>List of n-vertex “patch” objects. These topologies cannot be rendered directly – the tessellation units must be used to convert them into points, lines, or triangles to produce rasterization results. (VS, GS, and StreamOutput operations can also be performed.)</p>

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



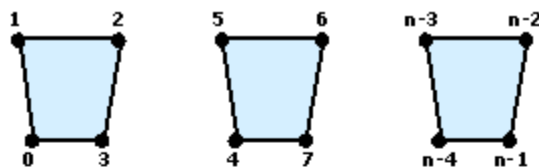
B6815-01

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

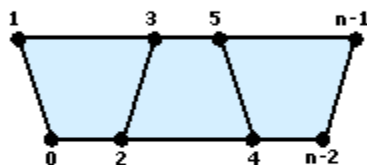


B6816-01

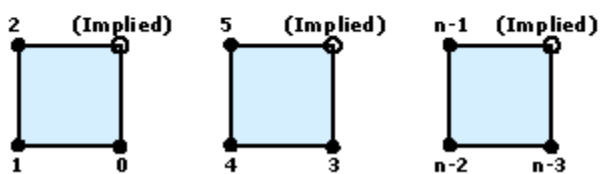
QUADLIST



QUADSTRIP



RECTLIST



B.6818-01

Vertex Data Overview

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

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

Vertex URB Entry (VUE) Formats

In general, vertex data is stored in Vertex URB Entries (VUEs) in the URB, 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	<p>Render Target Array Index (RTAIndex). This value is (eventually) used to index into a specific element of an "array" Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function.</p> <p>Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required, software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a reange check of computed index values against [0,2047], and output zero if that range is exceeded. Note that the unmodified "renderTargetArrayIndex" must be maintained in the VUE outside of the Vertex Header.</p> <p>Software can force an RTAIndex of 0 to be used (effectively ignoring the setting of this 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).</p> <p>Format: 0-based U32 index value</p>
D2	31:0	<p>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.</p> <p>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.</p> <p>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.</p> <p>Software can force a value of 0 to be used by programming Maximum VPIndex to 0.</p> <p>Format: 0-based U32 index value</p>
D3	31:0	<p>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.</p> <p>This field is read back by both the GS and Clip units.</p> <p>Format: FLOAT32</p>
D4	31:0	<p>Vertex Position X Coordinate. This field contains the X component of the vertex's 4D space position.</p> <p>Format: FLOAT32</p>
D5	31:0	<p>Vertex Position Y Coordinate. This field contains the Y component of the vertex's 4D space position.</p> <p>Format: FLOAT32</p>

DWord	Bits	Description
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	<p>ClipDistance 0 Value (optional). If the UserClipDistance Clip Test Enable Bitmask bit (3DSTATE_CLIP) is set, this value will be read from the URB in the Clip stage. If the value is found to be less than 0 or a NaN, the vertex's UCF<0> bit will set in the Clip unit's VertexClipTest function.</p> <p>If the UserClipDistance Clip Test Enable Bitmask bit is clear, this value will not be read back, and the vertex's UCF<0> bit will be zero by definition.</p> <p>Format: FLOAT32</p> <p>ClipDistance Values are enabled for clip/cull test in the Clip stage in one of two modes: Normally the corresponding Enable Bitmasks are obtained from the state programmed in the last "vertex-producing" stage (VS/DS/GS) that is enabled prior to the Clip stage. E.g., if VS and DS are enabled but GS is disabled, the masks are obtained from 3DSTATE_DS. Alternatively, the Enable Bitmasks can be obtained directly from corresponding masks programmed via 3DSTATE_CLIP, through use of 3DSTATE_CLIP's Force User Clip Distance [Cull/Clip] Test Enable Bitmask state bits (see description of 3DSTATE_CLIP).</p>
D9	31:0	ClipDistance 1 Value (optional). See above.
D10	31:0	ClipDistance 2 Value (optional). See above.
D11	31:0	ClipDistance 3 Value (optional). See above.
D12	31:0	ClipDistance 4 Value (optional). See above.
D13	31:0	ClipDistance 5 Value (optional). See above.
D14	31:0	ClipDistance 6 Value (optional). See above.
D15	31:0	ClipDistance 7 Value (optional). See above.
	31:0	<p>(Remainder of Vertex Elements).</p> <p>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.</p>

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.

In the CLIP unit, the read-back Position fields are interpreted as being in one of two coordinate systems, depending on the **CLIP_STATE.VertexPositionSpace** bit. The CLIP unit modifies its VertexClipTest function depending on the coordinate space of the incoming vertices.

VPOS_CLIPSPACE (Homogeneous 4D Clip-space coordinates, pre-perspective division): 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 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). When this coordinate system is selected, 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.

VPOS_SCREENSPACE (Screen Space position): Under certain circumstances, the position in the Vertex Header will contain the screen-space (pixel) coordinates (post viewport mapping).

The SF unit does not have a state bit defining the coordinate space of the incoming vertex positions. Software must use the Viewport Mapping function of the SF unit in order to ensure that screen-space coordinates are available after that function. If screen space coordinates are passed into SF, then software will likely turn off the Viewport Mapping function.

The following subsections briefly describe the three relevant coordinate spaces.

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

3DSTATE_VF

3DSTATE_VF_TOPOLOGY

Index Buffer (IB) State

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

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

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

Definition	Cut Index?
3DPRIM_POINTLIST	Y
3DPRIM_LINELIST	Y
3DPRIM_LINESTRIP	Y
3DPRIM_TRILIST	Y
3DPRIM_TRISTRIP	Y
3DPRIM_LINELIST_ADJ	Y
3DPRIM_LINESTRIP_ADJ	Y
3DPRIM_TRILIST_ADJ	Y
3DPRIM_TRISTRIP_ADJ	Y
3DPRIM_TRISTRIP_REVERSE	Y
3DPRIM_RECTLIST	N
3DPRIM_POINTLIST_BF	Y
3DPRIM_LINESTRIP_CONT	Y
3DPRIM_LINESTRIP_BF	Y
3DPRIM_LINESTRIP_CONT_BF	Y

Definition	Cut Index?
3DPRIM_TRIFAN_NOSTIPPLE	N

3DSTATE_INDEX_BUFFER

Vertex Buffers (VB) State

The 3DSTATE_VERTEX_BUFFERS and 3DSTATE_INSTANCE_STEP_RATE 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 VB's **BufferAccessType**. The VB's access type will determine whether the VF-computed VertexIndex or InstanceIndex is used to access data in the VB.

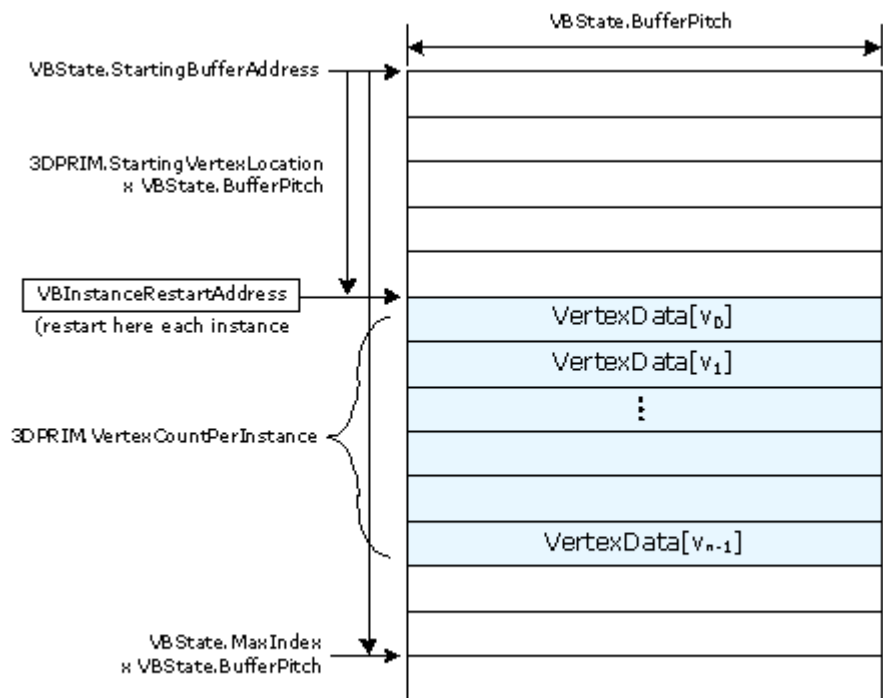
Given that the RANDOM access mode of the 3DPRIMITIVE command utilizes an IB (possibly provided by an application) to compute VB index values, VB definitions contain a **MaxIndex** value used to detect accesses beyond the end of the VBs. Any access outside the extent of a VB returns 0.

3DSTATE_VERTEX_BUFFERS

VERTEX_BUFFER_STATE

VERTEXDATA Buffers – SEQUENTIAL Access

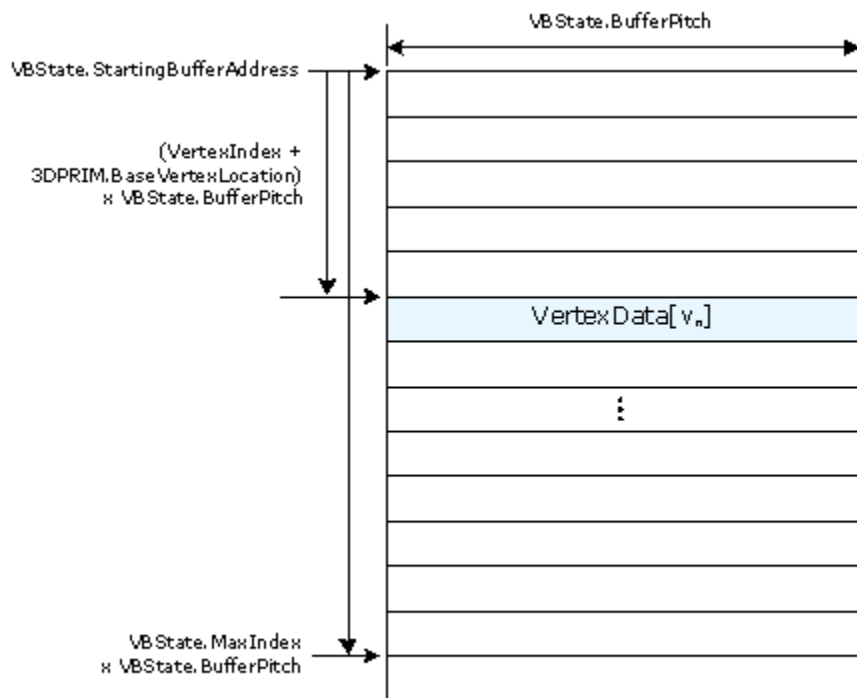
Description
<p>This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = SEQUENTIAL and (b) vertex elements with InstancingEnable set to DISABLED. Instead of “VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch”, the address of the byte immediately beyond the last valid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



B.6826-01

VERTEXDATA Buffers – RANDOM Access

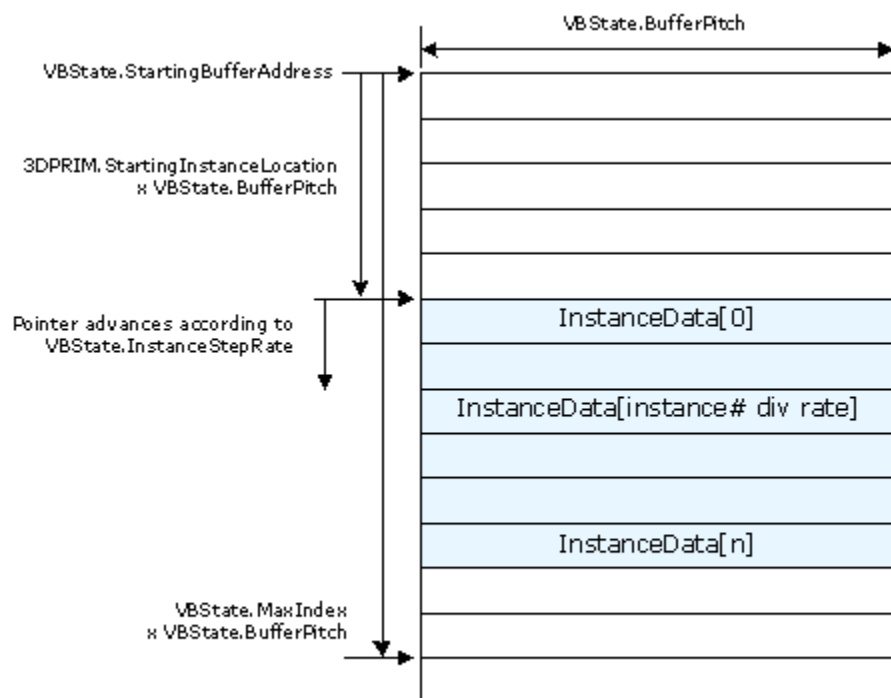
Description
<p>This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = RANDOM and (b) vertex elements with InstancingEnable set to DISABLED. Instead of “VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch”, the address of the byte immediately beyond the last valid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



B.6827-01

INSTANCEDATA Buffers

Description
<p>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”.</p>



B6839-01

Vertex Definition State

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

Input Vertex Definition

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

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

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

VERTEX_ELEMENT_STATE

3DSTATE_VERTEX_ELEMENTS

3D_Vertex_Component_Control

3DSTATE_VF_INSTANCING

3DSTATE_VF_SGVS

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.

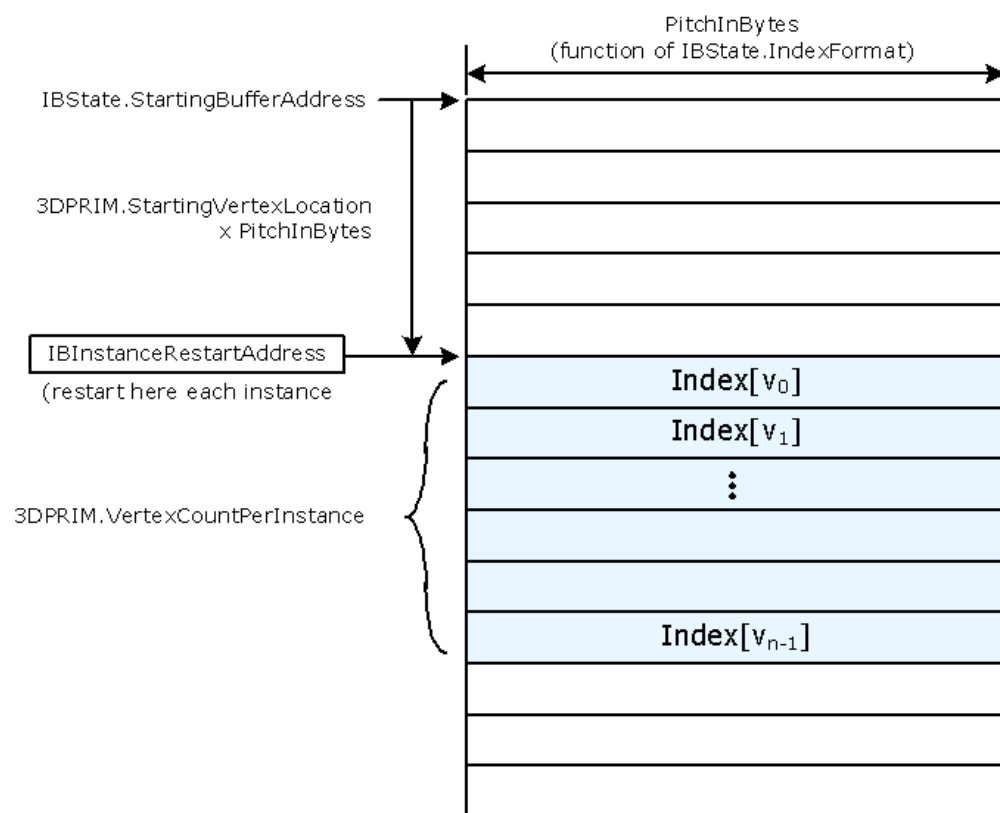
The optional insertion of VertexID and/or InstanceID into the input vertex data occurs as a separate step after the processing of VERTEX_ELEMENT structures, and is controlled via the 3DSTATE_VF_SGVS command.

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

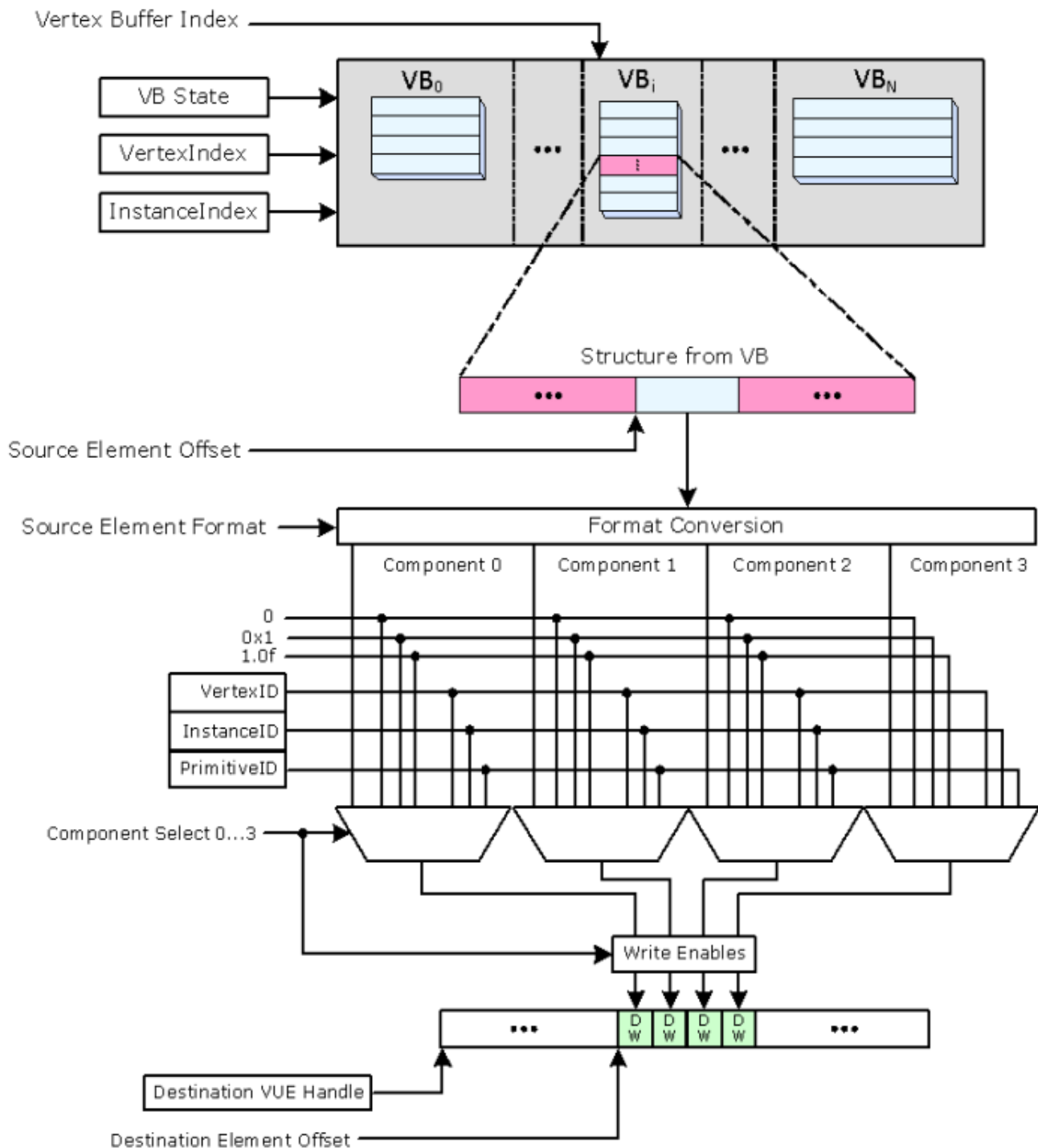
The following figure illustrates how the Index Buffer is accessed.



B6825-01

Vertex Element Data Path

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



B6840-01

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	-	-

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
R32G32_UINT	UINT	R	G	-	-
R32G32_UNORM	FLOAT	R	G	-	-
R32G32_SNORM	FLOAT	R	G	-	-
R64_FLOAT	FLOAT	R	-	-	-
R16G16B16A16_SSCELED	FLOAT	R	G	B	A
R16G16B16A16_USCELED	FLOAT	R	G	B	A
R32G32_SSCELED	FLOAT	R	G	-	-
R32G32_USCELED	FLOAT	R	G	-	-
R32G32_SFIED	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_USCELED	FLOAT	R	G	B	-
R8G8B8A8_SSCELED	FLOAT	R	G	B	A
R8G8B8A8_USCELED	FLOAT	R	G	B	A
R16G16_SSCELED	FLOAT	R	G	-	-
R16G16_USCELED	FLOAT	R	G	-	-

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
R32_SSCELED	FLOAT	R	-	-	-
R32_USCELED	FLOAT	R	-	-	-
R8G8_UNORM	FLOAT	R	G	-	-
R8G8_SNORM	FLOAT	R	G	-	-
R8G8_SINT	SINT	R	G	-	-
R8G8_UINT	UINT	R	G	-	-
R16_UNORM	FLOAT	R	-	-	-
R16_SNORM	FLOAT	R	-	-	-
R16_SINT	SINT	R	-	-	-
R16_UINT	UINT	R	-	-	-
R16_FLOAT	FLOAT	R	-	-	-
R8G8_SSCELED	FLOAT	R	G	-	-
R8G8_USCELED	FLOAT	R	G	-	-
R16_SSCELED	FLOAT	R	-	-	-
R16_USCELED	FLOAT	R	-	-	-
R8_UNORM	FLOAT	R	-	-	-
R8_SNORM	FLOAT	R	-	-	-
R8_SINT	SINT	R	-	-	-
R8_UINT	UINT	R	-	-	-
R8_SSCELED	FLOAT	R	-	-	-
R8_USCELED	FLOAT	R	-	-	-
R8G8B8_UNORM	FLOAT	R	G	B	-
R8G8B8_SNORM	FLOAT	R	G	B	-
R8G8B8_SSCELED	FLOAT	R	G	B	-
R8G8B8_USCELED	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_SSCELED	FLOAT	R	G	B	-
R16G16B16_USCELED	FLOAT	R	G	B	-
R16G16B16_UINT	UINT	R	G	B	-

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
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
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

3DSTATE_VS

3DSTATE_CONSTANT_VS

3DSTATE_PUSH_CONSTANT_ALLOC_VS

3DSTATE_BINDING_TABLE_POINTERS_VS

3DSTATE_SAMPLER_STATE_POINTERS_VS

3DSTATE_URB_VS

Functions

Vertex Shader Cache (VS\$)

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

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

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

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

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

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

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

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

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

SIMD4x2 VS Thread Request Generation

Description
BDW adds SIMD8 VS thread request generation, which is described in a following section.

The following discussion assumes the VS Function is ENABLED.

When the Vertex Cache is disabled, the VS unit passes each pair of incoming vertices to a VS thread. Under certain circumstances (e.g., prior to a state change or pipeline flush) the VS unit spawns a VS thread to process a single vertex. Note that, in this case, the “unused” vertex slot is “disabled” via the Execution Mask provided by the VS unit to the GEN4 subsystem as part of the thread dispatch (See the EU ISA volume). The VS thread is itself unaware of the single-vertex case, and therefore a single VS kernel can be used to process one or two vertices. (The performance of single-vertex processing roughly equals the two-vertex case.)

When the Vertex Cache is enabled, the VF unit detects vertices that hit in the cache and marks these vertices so that they bypass VS thread processing and are output via a reference to the cached VUE. The VS unit keeps track of these cache-hit vertices as it proceeds to process cache-miss vertices. The VS unit guarantees that vertices exit the unit in the order they are received. This may require the VS unit to issue single-vertex VS threads to process a cache-miss vertex that has yet to be paired up with another cache-miss vertex (if this condition is preventing the VS unit from producing any output).

SIMD4x2 VS Thread Execution

Description
BDW adds SIMD8 VS thread execution, which is described in a following section.

A VS kernel (with one exception mentioned below) assumes it is to operate on two vertices in parallel. Input data is either passed directly in the thread payload (including the input vertex data) or indirectly via pointers passed in the payload.

Refer to the *EU ISA* chapters for specifics on writing kernels that operate in SIMD4x2 fashion.

Refer to the 3D Pipeline Stage Overview (*3D Overview*) for information on FF-unit/thread interactions.

In the (unlikely) event that the VS kernel needs to determine whether it is processing one or two vertices, the kernel can compare the **URB Return Handle 0** and **URB Return Handle 1** fields of the thread payload. These fields differ if two vertices are being processed, and identical if one vertex is being processed. An example of when this test may be required is if the kernel outputs some vertex-dependent results into a memory buffer; without the test the single vertex case might incorrectly output two sets of results. Note that this is not the case for writing the URB destinations, as the Execution Mask prevents the write of an undefined output.

VS Thread Dispatch Masks

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

SIMD4x2 Dispatch Mask

In SIMD4x2 dispatch mode, the EU Dispatch Mask is initialized as a function of the number of vertices included in the thread dispatch, as follows:

- 1 vertex: 0x0000000F
- 2 vertices: 0x000000FF

SIMD8 Dispatch Mask

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

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

Vertex Output

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

Thread Termination

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

Primitive Output

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

Statistics Gathering

The VS stage tracks a single pipeline statistic, the number of times a vertex shader is executed. A vertex shader is executed for each vertex that is fetched on behalf of a 3DPRIMITIVE command, unless the shaded results for that vertex are already available in the vertex cache. If the **Statistics Enable** bit in VS_STATE is set, the VS_INVOCATION_COUNT Register (see Memory Interface Registers in Volume Ia, GPU) will be incremented for *each vertex* that is dispatched to a VS thread. This counter will often need to be incremented by 2 for each thread invoked since 2 vertices are dispatched to one VS thread in the general case.

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

Payloads

SIMD4x2 Payload

The following table describes the payload delivered to VS threads.

VS Thread Payload (SIMD4x2)

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 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 <i>3D Pipeline</i> 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 Range: 0-503
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 <i>3D Pipeline</i> 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	<p>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.</p> <p>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 <i>Generic FF Unit</i> for further description).</p> <p>Format:</p> <p>U14 64B-aligned URB offset.</p>
R0.0	31:16	Reserved
	15:0	<p>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.</p> <p>(See <i>Generic FF Unit</i> for further description).</p> <p>Format:</p> <p>U14 64B-aligned URB offset.</p>
[Varies] optional	255:0	<p>Constant Data (optional):</p> <p>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_VS command (taking the buffer enables into account).</p> <p>The Constant Data arrives in a non-interleaved format.</p>
Varies	255:0	<p>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.</p> <p>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.</p>

SIMD8 Payload

The following table describes the payload delivered to VS threads.

SIMD8 VS Thread Payload

DWord	Bits	Description
R0.7	31:0	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 <i>3D Pipeline</i> 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: Range: 0-503
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). (See <i>3D Pipeline</i> 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
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)

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

Hull Shader (HS) Stage

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

Programming Note	
Context:	Hull Shader (HS) Stage
For SW Tessellation mode, the HS thread can also write tessellated domain point topologies to memory. The domain point count and starting memory address of the domain points are passed via the Patch Header in the patch record.	

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.

3DSTATE_HS

3DSTATE_PUSH_CONSTANT_ALLOC_HS

3DSTATE_CONSTANT_HS

3DSTATE_CONSTANT(Body)

3DSTATE_BINDING_TABLE_POINTERS_HS

3DSTATE_SAMPLER_STATE_POINTERS_HS

3DSTATE_URB_HS

Functions

Patch Object Staging

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

HS Thread Execution

Input to HS threads is comprised of:

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

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

HS Thread Dispatch Mask

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

SINGLE_PATCH Dispatch Mask

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

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).

In SW Tessellation mode, the header contains **Domain Point Count** and **Domain Point Buffer Starting Address** fields, which identify the domain points generated by an HS thread.

The following table shows the fixed layouts of the Patch Header DW0-7, depending on the SW Tessellation Mode. Also see the fixed layouts depending on DomainType that follow.

Patch Header (SW Tessellation Mode)

DWord	Bits	Description
7	31:0	Domain Point Count Specifies the number of DOMAIN_POINT structures in the domain point list in memory. If 0, there are no domain points defined, the patch will considered "culled", and the TE stage will discard the patch. Otherwise the TS stage will send this number of domain points down the pipeline. Format: U32
6	31:6	Domain Point Buffer Starting Address (DPBSA) This field specifies the starting memory offset from SW Tessellation Base Address (set by the SWTESS_BASE_ADDRESS command) at which the HS thread has written a list of DOMAIN_POINT structures. This field is ignored if Domain Point Count is 0. Format: 64B-aligned offset from SW Tessellation Base Address
	5:0	Reserved: MBZ
5-0	31:0	Reserved: MBZ

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
1	31:0	Reserved : MBZ
0	31:1	Reserved : MBZ
	0	<div> <div>Description</div> <div> <p>TR DS Cache Disable</p> <p>Setting this bit will cause the DS Cache to be disabled for tessellator-output topologies resulting from the tessellation of stitch-transition regions. Note that the DS Cache will remain enabled for other topologies generated for the patch (i.e., the inner region if one exists, or the entire tessellated patch if there are no transition regions). The HS kernel shall clear this bit when the patch output topology is POINT.</p> <p>While disabling the DS Cache usually results in reduced performance, when domain points hit in this cache more than 11 times per patch a more significant performance penalty may be incurred. Therefore it is recommended that the HS kernel include an algorithm to set this bit when such conditions are met. A more conservative algorithm (i.e., setting the bit when conditions are close to being met) can be employed in order to limit the execution overhead of this algorithm. Such an algorithm could possibly examine the ratio of each outer edge's tessellation factor to the corresponding inner tessellation factor, thus determining the (possibly-approximated) count of triangles including any given domain point in that transition region. If any of the transition region domain points are included in more than 11 triangles (or may approach that number), the bit should be set for the patch. Note that some domain points (such as the corners of the inside region) can be connected to two adjacent transition regions, therefore cache hits from both regions must be considered. Other exceptional cases include when the inner region of patch with EVEN or INTEGER partitioning collapses to a single point, in which case that central domain point is included in every tessellated triangle.</p> </div> </div>

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
		TR DS Cache Disable See description of the corresponding bit in the QUAD domain patch header (above).

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

HW Bug: The Tessellation stage will incorrectly add domain points along patch edges under the following conditions, which may result in conformance failures and/or cracking artifacts:

- QUAD domain
- INTEGER partitioning
- All three TessFactors in a given U or V direction (e.g., V direction: UEQ0, InsideV, UEQ1) are all exactly 1.0
- All three TessFactors in the other direction are > 1.0 and all round up to the same integer value (e.g, U direction: VEQ0 = 3.1, InsideU = 3.7, VEQ1 = 3.4)

The suggested workaround (to be implemented as part of the postamble to the HS shader in the HS kernel) is:

```
if (
    (TF[UEQ0] > 1.0) ||
    (TF[VEQ0] > 1.0) ||
    (TF[UEQ1] > 1.0) ||
    (TF[VEQ1] > 1.0) ||
    (TF[INSIDE_U] > 1.0) ||
    (TF[INSIDE_V] > 1.0) )
{
    TF[INSIDE_U] = (TF[INSIDE_U] == 1.0) ? 2.0 : TF[INSIDE_U];
    TF[INSIDE_V] = (TF[INSIDE_V] == 1.0) ? 2.0 : TF[INSIDE_V];
}
```

DOMAIN_POINT Structure

In SW Tessellation Mode (i.e., when the TE State is SW_TESS), the TE stage reads a sequence of DOMAIN_POINT structures from memory, starting at the Domain Point Buffer Starting Address field of the patch header. (The DPBSA is treated as an offset from the SW Tessellation Base Address as set by the SWTESS_BASE_ADDRESS command.)

DOMAIN_POINT Memory Structure (SW Tessellation)

DWord	Bits	Description
0	31	PrimStart Set on the first domain point of the topology (e.g., first vertex in a TRISTRIP).
	30	PrimEnd Set on the last domain point of the topology (e.g., last vertex in a TRISTRIP). Programming note: Software must ensure that incomplete primitives are not output, or behavior is UNDEFINED.
	29	PatchEnd Set on the last domain point for the <i>patch</i> . By definition, PrimEnd must also be set. Programming Note: Software must ensure that the <i>Domain Point Count</i> coincides with the domain point marked with PatchEnd.
	28:24	PrimType This is the primitive topology type. Format: See 3DPRIMITIVE for encodings Valid values: POINTLIST, LINESTRIP, LINELIST, TRISTRIP, TRISTRIP_REV, TRILIST, TRIFAN.
	23:19	Reserved
	18:17	DS Tag [16:15] This field provides bits [16:15] of the DS Tag value for this domain point. See DS Tag [14:0] Format: U2
	16:0	U Coordinate Format: U1.16
1	31:17	DS Tag [14:0] This field provides bits [14:0] of the DS Tag value for this domain point. In order to utilize the DS cache, the 17-bit DS Tag must be unique for the associated U,V coordinate. If software cannot guarantee this, the DS cache must be disabled when in SW Tessellation mode. Format: U15
	16:0	V Coordinate Format: U1.16

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	31:0	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: U9 Range: 0-503
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.

GRF DWord	Bits	Description
	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: U14 64B-aligned URB offset.
R1 is only included for dispatches that have Include Vertex Handles enabled.		
R1.7	31:16	ICP 7 Handle ID
	15:0	ICP 7 Handle Format: U14 64B-aligned URB offset.
R1.6	31:16	ICP 6 Handle ID
	15:0	ICP 6 Handle
R1.5	31:16	ICP 5 Handle ID
	15:0	ICP 5 Handle
R1.4	31:16	ICP 4 Handle ID
	15:0	ICP 4 Handle
R1.3	31:16	ICP 3 Handle ID
	15:0	ICP 3 Handle
R1.2	31:16	ICP 2 Handle ID
	15:0	ICP 2 Handle
R1.1	31:16	ICP 1 Handle ID
	15:0	ICP 1 Handle
R1.0	31:16	ICP 0 Handle ID
	15:0	ICP 0 Handle

GRF DWord	Bits	Description
R2 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count > 7		
R2.7	31:16	ICP 15 Handle ID
	15:0	ICP 15 Handle
R2.6	31:16	ICP 14 Handle ID
	15:0	ICP 14 Handle
R2.5	31:16	ICP 13 Handle ID
	15:0	ICP 13 Handle
R2.4	31:16	ICP 12 Handle ID
	15:0	ICP 12 Handle
R2.3	31:16	ICP 11 Handle ID
	15:0	ICP 11 Handle
R2.2	31:16	ICP 10 Handle ID
	15:0	ICP 10 Handle
R2.1	31:16	ICP 9 Handle ID
	15:0	ICP 9 Handle
R2.0	31:16	ICP 8 Handle ID
	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	ICP 23 Handle ID
	15:0	ICP 23 Handle
R3.6	31:16	ICP 22 Handle ID
	15:0	ICP 22 Handle
R3.5	31:16	ICP 21 Handle ID
	15:0	ICP 21 Handle
R3.4	31:16	ICP 20 Handle ID
	15:0	ICP 20 Handle
R3.3	31:16	ICP 19 Handle ID
	15:0	ICP 19 Handle
R3.2	31:16	ICP 18 Handle ID
	15:0	ICP 18 Handle
R3.1	31:16	ICP 17 Handle ID
	15:0	ICP 17 Handle
R3.0	31:16	ICP 16 Handle ID
	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	ICP 31 Handle ID

GRF DWord	Bits	Description
	15:0	ICP 31 Handle
R4.6	31:16	ICP 30 Handle ID
	15:0	ICP 30 Handle
R4.5	31:16	ICP 29 Handle ID
	15:0	ICP 29 Handle
R4.4	31:16	ICP 28 Handle ID
	15:0	ICP 28 Handle
R4.3	31:16	ICP 27 Handle ID
	15:0	ICP 27 Handle
R4.2	31:16	ICP 26 Handle ID
	15:0	ICP 26 Handle
R4.1	31:16	ICP 25 Handle ID
	15:0	ICP 25 Handle
R4.0	31:16	ICP 24 Handle ID
	15:0	ICP 24 Handle
[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_HS command (taking the buffer enables into account).
[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.

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.

Description
The TE stage can also operate in SW Tessellation mode, where it simply reads “pre-tessellated” domain point topologies from memory and passes them down the pipeline.

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.

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, 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.

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.

Description
In SW Tessellation mode, a Domain Point Count of 0 indicates that a patch is to be culled.

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).

Usage Note: Except for the purposes of debug or other experimentation, these max TessFactor values shall be programmed to values required by the APIs (refer to the 3DSTATE_TE definition).

Partitioning

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

- **INTEGER:** The edge is divided into an integral number of equal segments (given some fixed-point tolerance).

After clamping, the TF is rounded up to an integer value. The edge is divided into that many equal segments.

- **EVEN_FRACTIONAL:** The edge is divided into an *even* number of possibly-unequal segments. The total number of segments is determined by rounding up the post-clamped TF to an even number.

More specifically, the edge is divided exactly in half. Like the endpoints of the edge, the midpoint of the edge is by definition a tessellation point. Each half contains some number of equal segments and possibly one smaller segment. The size of the smaller segment is determined by the position of the TF value within the range defined by the TF rounded down and up to even numbers. The closer the TF is to the smaller value, the smaller the segment size is. When the TF reaches the smaller even value, the smaller segment disappears. The closer the TF gets to the larger even value, the closer the smaller segment size approaches the size of the other segments. When the TF reaches the larger even value, all segments are equal. The position of the smaller segment along the half edge varies as a function of the TF value.

- **ODD_FRACTIONAL:** The edge is divided into an *odd* number of possibly-unequal segments. The tessellation scheme is very similar to EVEN_FRACTIONAL partitioning, except that the edge midpoint is not included as a tessellation point. This, and the fact that the tessellation points are mirrored about the edge midpoint, causes an "odd" segment (which may or may not be the

"smaller" segment) to straddle the edge midpoint, therefore resulting in the number of segments for the edge always being odd.

Domain Types and Output Topologies

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

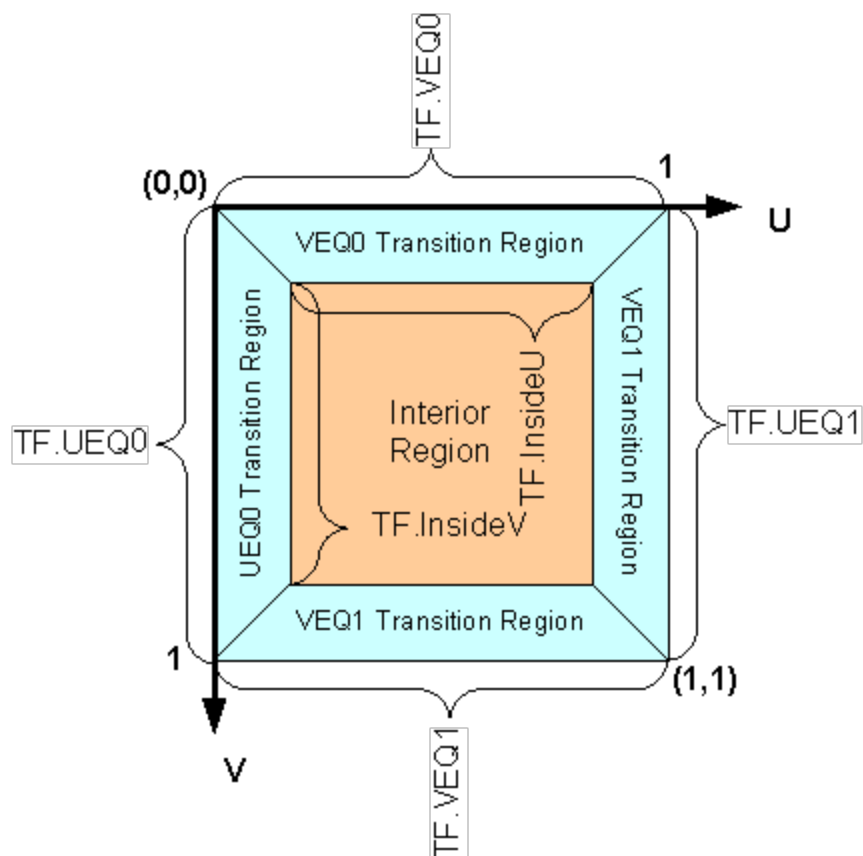
- **QUAD:** A square 2D region within a u,v Cartesian (rectangular) space. The region extends from the origin to $u=1$ and $v=1$. Within the region, tessellation domain locations are determined. The possible output topologies include points, clockwise triangles, and counter-clockwise triangles.
- **TRI:**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

QUAD Domain Tessellation

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

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

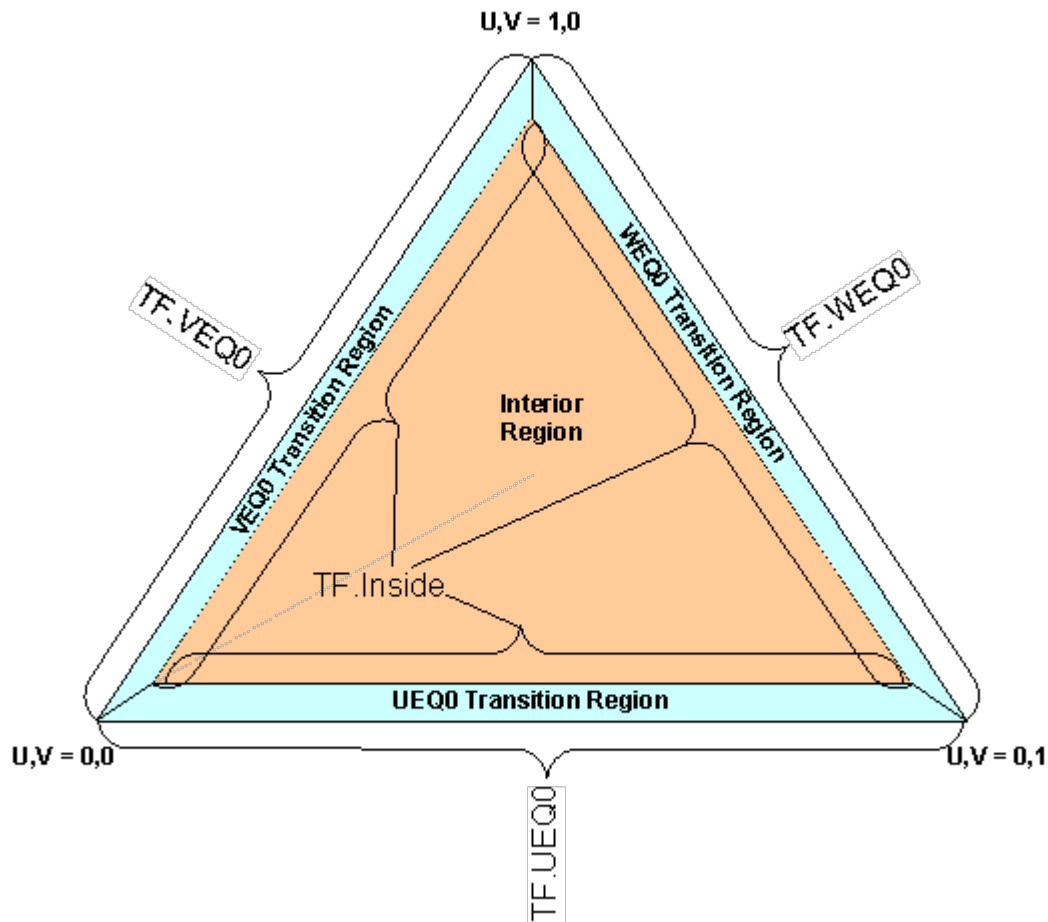
QUAD Domain



TRI Domain Tessellation

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

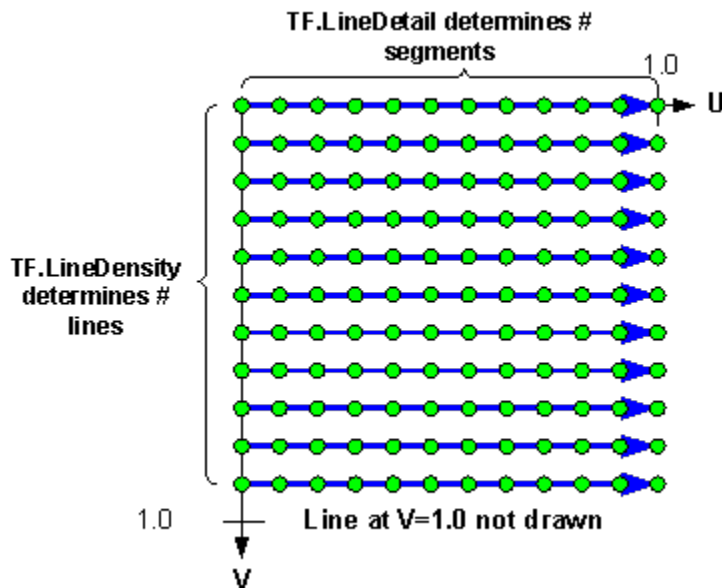
TRI Domain



ISOLINE Domain Tessellation

Tessellation of the ISOLINE domain is different but much simpler than QUAD and TRI domains. The TF.LineDetail TF controls how finely the U direction is tessellated, while the TF.LineDensity TF controls how finely the V direction is tessellated. When LINE output topology is selected, a series of segmented lines parallel to the U axis (constant V) are output. When POINT output topology is selected, only the line segment endpoints are output (as point objects). In either case there is no topology output for the $V=1.0$ 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

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.

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

SIMD4x2 Payload

The following table describes the payload delivered to DS threads.

DS Thread Payload (SIMD4x2)

DWord	Bits	Description
R0.7	31:0	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 <i>3D Pipeline</i> 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
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	31:26	Reserved

DWord	Bits	Description		
	25:16	Handle ID 1. This ID is assigned by the FF unit and used to identify the URB Return Handle 1 to the FF unit (as FF-specific index value, not a URB address). 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 = Reserved for HW Implementation Use.		
	15:14	Reserved		
	13:0	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>	Format	U14 handle (512-bit granular)
Format				
U14 handle (512-bit granular)				
R0.0	31:26	Reserved		
	25:16	<table><tr><th>Description</th></tr><tr><td>Handle ID 0. This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). Format = Reserved for HW Implementation Use.</td></tr></table>	Description	Handle ID 0. This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). Format = Reserved for HW Implementation Use.
		Description		
	Handle ID 0. This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). Format = Reserved for HW Implementation Use.			
15:14	Reserved			
13:0	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>	Format	U14 handle (512-bit granular)	
Format				
U14 handle (512-bit granular)				
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).		

DWord	Bits	Description
		Format: <div><div>Format</div><div>U14 handle)</div></div>
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.
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	31:0	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 <i>3D Pipeline</i> 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
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

DWord	Bits	Description
	26:16	Patch Handle ID. This ID is assigned by the FF unit and used to identify the patch URB entry to the FF unit (as FF-specific index value, not a URB address). Format = Reserved for HW Implementation Use.
	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.)
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.)

DWord	Bits	Description
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). 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).
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).

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.

For BDW, 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	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	
POINTLIST	[0] = (<u>0</u>); [1] = (<u>1</u>); ...; [N-2] = (<u>N-2</u>);	
POINTLIST_BF	N/A	
LINELIST (N is multiple of 2)	[0] = (0, <u>1</u>); [1] = (2, <u>3</u>); ...; [(N/2)-1] = (N-2, <u>N-1</u>)	
LINELIST_ADJ (N is multiple of 4)	[0] = (0,1, <u>2</u> ,3); [1] = (4,5, <u>6</u> ,7); ...; [(N/4)-1] = (N-4,N-3, <u>N-2</u> ,N-1)	
LINESTRIP (N >= 2)	[0] = (0, <u>1</u>); [1] = (1, <u>2</u>); ...; [N-2] = (N-2, <u>N-1</u>)	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT (N >= 4)	[0] = (0,1, <u>2</u> ,3); [1] = (1,2, <u>3</u> ,4); ...; [N-4] = (N-4,N-3, <u>N-2</u> ,N-1)	LINESTRIP_ADJ_CONT is added 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, <u>1</u>); [1] = (1, <u>2</u>); [N] = (N-1, <u>0</u>);	Not supported after GS.

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST	Same as TRILIST	Handled same as TRILIST
TRILIST_ADJ (N is multiple of 6)	[0] = (0,1,2,3,4,5); [1] = (6,7,8,9,10,11); ...; [(N/6)-1] = (N-6,N-5,N-4,N-3,N-2,N-1)	
TRISTRIP (Reorder Leading) (N >= 3)	[0] = (0,1,2); {TRISTRIP} [1] = (1,3,2); {TRISTRIP_REV} [k even] = (k,k+1,k+2) {TRISTRIP} [k odd] = (k,k+2,k+1) {TRISTRIP_REV} [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.
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.

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	
TRISTRIP_ADJ (<u>Reorder Leading</u>) (N >= 6)	N = 6 or 7: [0] = (0,1,2,5, <u>4</u> ,3) N = 8 or 9: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (2,5, <u>6</u> ,7,4,0); ...; N >= 10: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (2,5, <u>6</u> ,8,4,0); ...; [k>1, even] = (2k,2k-2, 2k+2, 2k+6, <u>2k+4</u> , 2k+3); [k>2, odd] = (2k, 2k+3, <u>2k+4</u> , 2k+6, 2k+2, 2k-2);...; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1, <u>N-2</u> ,N-3); [(N/2)-3, odd] = (N-6,N-3, <u>N-2</u> ,N-1,N-4,N-8);	Objects have vertices reordered.
TRISTRIP_ADJ (<u>Reorder Trailing</u>) (N >= 6)	N = 6 or 7: [0] = (0,1,2,5, <u>4</u> ,3) N = 8 or 9: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (4,0,2,5, <u>6</u> ,7); ...; N >= 10: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (4,0,2,5, <u>6</u> ,8); ...; [k>1, even] = (2k,2k-2, 2k+2, 2k+6, <u>2k+4</u> , 2k+3); [k>2, odd] = (2k+2, 2k-2, 2k, 2k+3, <u>2k+4</u> , 2k+6);...; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1, <u>N-2</u> ,N-3); [(N/2)-3, odd] = (N-4,N-8,N-6,N-3, <u>N-2</u> ,N-1);	OpenGL ordering rules (last non-adjacent vertex is the last – aka provoking – vertex of the triangle). Even triangles have the same ordering as Leading Vertex, odd triangle ordering is different (rotated 2 vertices).
TRIFAN (N > 2)	[0] = (0,1, <u>2</u>); [1] = (0,2, <u>3</u>); ...; [N-3] = (0, N-2, <u>N-1</u>);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is added for BDW. POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	
	[0] = (0,1,2, <u>3</u>); [1] = (4,5,6, <u>7</u>); ...; [(N/4)-1] = (N-4,N-3,N-2, <u>N-1</u>);	Not supported after GS. : QUADLIST primitives are converted into POLYGONS in VF, and therefore never reach the GS.
	[0] = (0,1,3, <u>2</u>); [1] = (2,3,5, <u>4</u>); ...; [(N/2)-2] = (N-4,N-3,N-1, <u>N-2</u>);	Not supported after GS. : QUADSTRIP primitives are converted into POLYGONS in VF, and therefore never reach the GS.
BDW: PATCHLIST_1 PATCHLIST_2 PATCHLIST_3..32	[0] = (<u>0</u>); [1] = (<u>1</u>); ...; [N-2] = (<u>N-2</u>); [0] = (0, <u>1</u>); [1] = (2, <u>3</u>); ...; [(N/2)-1] = (N-2, <u>N-1</u>) 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.

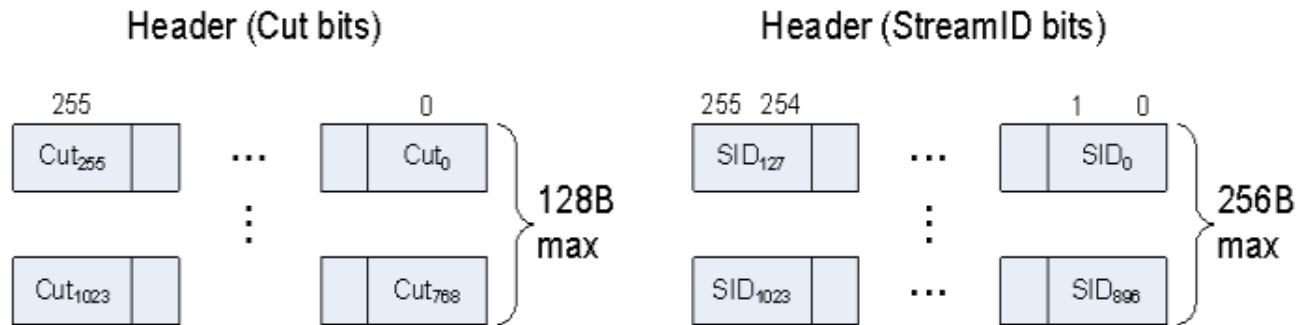
The primary usage models for GS threads include (possible combinations of):

- 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 used to emulate API functions not supported by specialized hardware. These functions might include (but are not limited to):
 - Conversion of API-defined topologies into topologies that can be rendered (e.g., LINELOOP → LINESTRIP, POLYGON → TRIANGLE, QUAD → 2 TRIANGLES, 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.

GS URB Entry

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



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

- When rendering is DISABLED, typically output vertices are 32B-aligned, with the exception of 16B-alignment for vertices $\leq 16B$ in length.
 - The absolute worst case size comes from three DW scalars output per vertex. If these are, say, three ".x" outputs, you need to store each DW in a 128b (16B) element, plus another pad 16B to keep the 32B alignment. So you require $4 \times 16B = 64B/\text{vertex}$. You have to have room for $1024 \text{ scalars} / 3 \text{ scalar/vtx} = 341 \text{ vertices}$. $341 \times 64B = 21,824B$. Then add 96B to hold 2b/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 $1024 \text{ scalars} / 5 = 204 \text{ vertices}$. $204 \times 64 = 13,056B$. 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

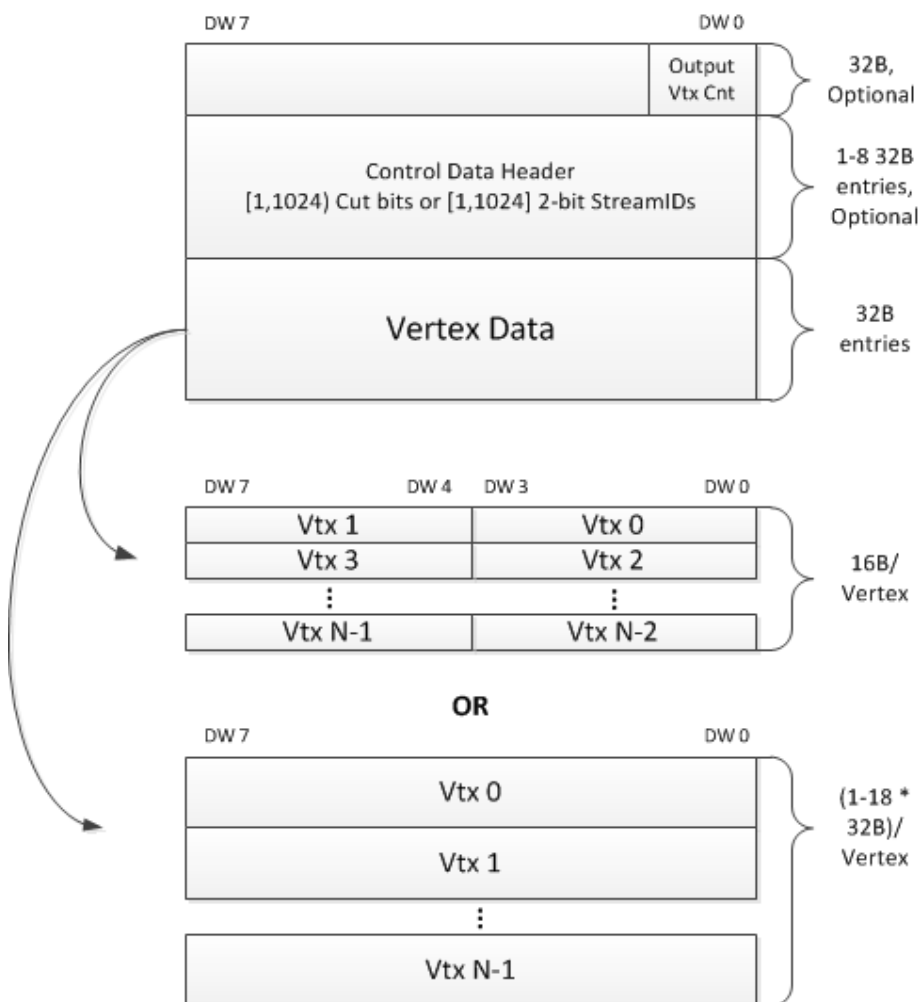
The GS URB entry is the same as in the two previous generations with the following exception: If **Static Output** (3DSTATE_GS) is clear, the URB entry starts with a 32B OUTPUT_VERTEX_COUNT structure as defined below. This control header (if present) immediately follows this structure. If **Static Output**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

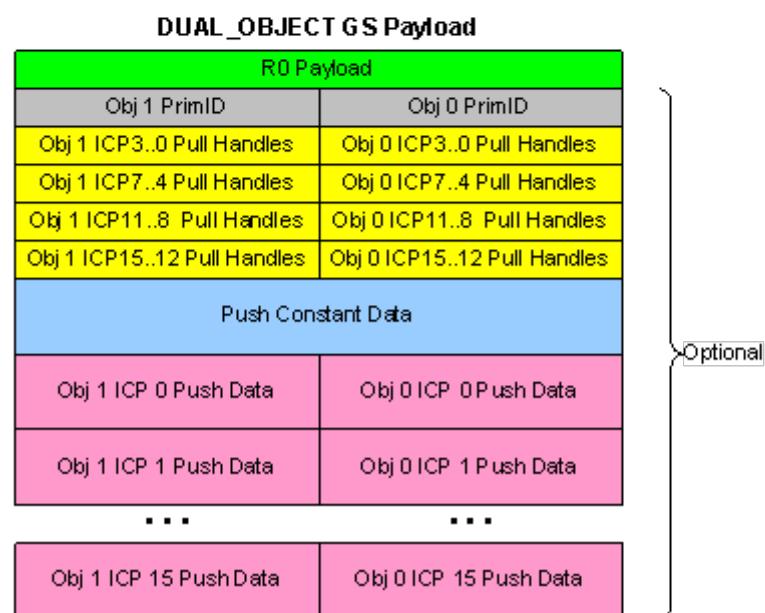
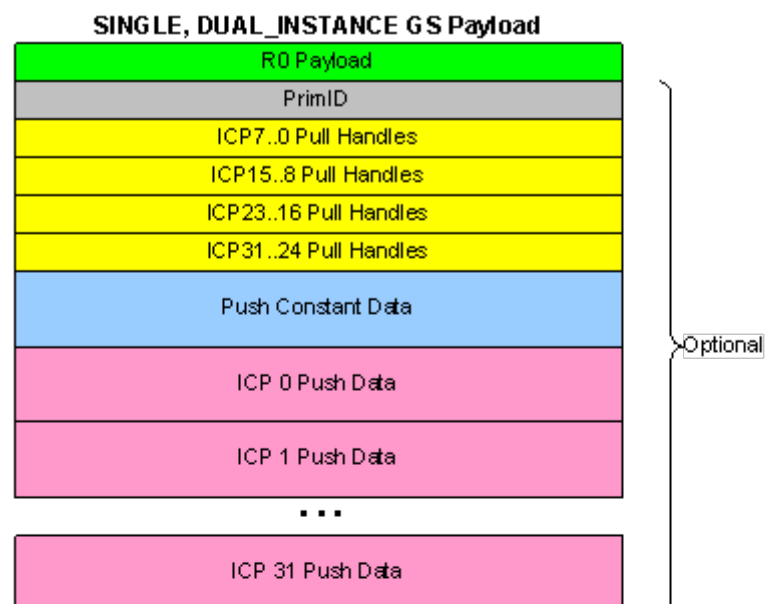
Refer to the *Statistics Gathering* summary provided earlier in this specification. Refer to the *Memory Interface Registers* chapter for details on these MMIO pipeline statistics counter registers, as well as the chapters corresponding to the other functions involved (e.g., DataPort, URB shared functions).

Payloads

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	31:0	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
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 <i>3D Pipeline</i> .
	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	ICP 7 Handle ID
	15:0	ICP 7 Handle

GRF DWord	Bits	Description
Rn.6	31:16	ICP 6 Handle ID
	15:0	ICP 6 Handle
Rn.5	31:16	ICP 5 Handle ID
	15:0	ICP 5 Handle
Rn.4	31:16	ICP 4 Handle ID
	15:0	ICP 4 Handle
Rn.3	31:16	ICP 3 Handle ID
	15:0	ICP 3 Handle
Rn.2	31:16	ICP 2 Handle ID
	15:0	ICP 2 Handle
Rn.1	31:16	ICP 1 Handle ID
	15:0	ICP 1 Handle
Rn.0	31:16	ICP 0 Handle ID
	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	ICP 15 Handle ID
	15:0	ICP 15 Handle
Rn+1.6	31:16	ICP 14 Handle ID
	15:0	ICP 14 Handle
Rn+1.5	31:16	ICP 13 Handle ID
	15:0	ICP 13 Handle
Rn+1.4	31:16	ICP 12 Handle ID
	15:0	ICP 12 Handle
Rn+1.3	31:16	ICP 11 Handle ID
	15:0	ICP 11 Handle
Rn+1.2	31:16	ICP 10 Handle ID
	15:0	ICP 10 Handle
Rn+1.1	31:16	ICP 9 Handle ID
	15:0	ICP 9 Handle
Rn+1.0	31:16	ICP 8 Handle ID
	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	ICP 23 Handle ID
	15:0	ICP 23 Handle

GRF DWord	Bits	Description
Rn+2.6	31:16	ICP 22 Handle ID
	15:0	ICP 22 Handle
Rn+2.5	31:16	ICP 21 Handle ID
	15:0	ICP 21 Handle
Rn+2.4	31:16	ICP 20 Handle ID
	15:0	ICP 20 Handle
Rn+2.3	31:16	ICP 19 Handle ID
	15:0	ICP 19 Handle
Rn+2.2	31:16	ICP 18 Handle ID
	15:0	ICP 18 Handle
Rn+2.1	31:16	ICP 17 Handle ID
	15:0	ICP 17 Handle
Rn+2.0	31:16	ICP 16 Handle ID
	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	ICP 31 Handle ID
	15:0	ICP 31 Handle
Rn+3.6	31:16	ICP 30 Handle ID
	15:0	ICP 30 Handle
Rn+3.5	31:16	ICP 29 Handle ID
	15:0	ICP 29 Handle
Rn+3.4	31:16	ICP 28 Handle ID
	15:0	ICP 28 Handle
Rn+3.3	31:16	ICP 27 Handle ID
	15:0	ICP 27 Handle
Rn+3.2	31:16	ICP 26 Handle ID
	15:0	ICP 26 Handle
Rn+3.1	31:16	ICP 25 Handle ID
	15:0	ICP 25 Handle
Rn+3.0	31:16	ICP 24 Handle ID
	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	Object 1 ICP 3 Handle ID
	15:0	Object 1 ICP 3 Handle
Rn.6	31:16	Object 1 ICP 2 Handle ID

GRF DWord	Bits	Description
	15:0	Object 1 ICP 2 Handle
Rn.5	31:16	Object 1 ICP 1 Handle ID
	15:0	Object 1 ICP 1 Handle
Rn.4	31:16	Object 1 ICP 0 Handle ID
	15:0	Object 1 ICP 0 Handle
Rn.3	31:16	Object 0 ICP 3 Handle ID
	15:0	Object 0 ICP 3 Handle
Rn.2	31:16	Object 0 ICP 2 Handle ID
	15:0	Object 0 ICP 2 Handle
Rn.1	31:16	Object 0 ICP 1 Handle ID
	15:0	Object 0 ICP 1 Handle
Rn.0	31:16	Object 0 ICP 0 Handle ID
	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	Object 1 ICP 7 Handle ID
	15:0	Object 1 ICP 7 Handle
Rn+1.6	31:16	Object 1 ICP 6 Handle ID
	15:0	Object 1 ICP 6 Handle
Rn+1.5	31:16	Object 1 ICP 5 Handle ID
	15:0	Object 1 ICP 5 Handle
Rn+1.4	31:16	Object 1 ICP 4 Handle ID
	15:0	Object 1 ICP 4 Handle
Rn+1.3	31:16	Object 0 ICP 7 Handle ID
	15:0	Object 0 ICP 7 Handle
Rn+1.2	31:16	Object 0 ICP 6 Handle ID
	15:0	Object 0 ICP 6 Handle
Rn+1.1	31:16	Object 0 ICP 5 Handle ID
	15:0	Object 0 ICP 5 Handle
Rn+1.0	31:16	Object 0 ICP 4 Handle ID
	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	Object 1 ICP 11 Handle ID
	15:0	Object 1 ICP 11 Handle
Rn+2.6	31:16	Object 1 ICP 10 Handle ID

GRF DWord	Bits	Description
	15:0	Object 1 ICP 10 Handle
Rn+2.5	31:16	Object 1 ICP 9 Handle ID
	15:0	Object 1 ICP 9 Handle
Rn+2.4	31:16	Object 1 ICP 8 Handle ID
	15:0	Object 1 ICP 8 Handle
Rn+2.3	31:16	Object 0 ICP 11 Handle ID
	15:0	Object 0 ICP 11 Handle
Rn+2.2	31:16	Object 0 ICP 10 Handle ID
	15:0	Object 0 ICP 10 Handle
Rn+2.1	31:16	Object 0 ICP 9 Handle ID
	15:0	Object 0 ICP 9 Handle
Rn+2.0	31:16	Object 0 ICP 8 Handle ID
	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	Object 1 ICP 15 Handle ID
	15:0	Object 1 ICP 15 Handle
Rn+3.6	31:16	Object 1 ICP 14 Handle ID
	15:0	Object 1 ICP 14 Handle
Rn+3.5	31:16	Object 1 ICP 13 Handle ID
	15:0	Object 1 ICP 13 Handle
Rn+3.4	31:16	Object 1 ICP 12 Handle ID
	15:0	Object 1 ICP 12 Handle
Rn+3.3	31:16	Object 0 ICP 15 Handle ID
	15:0	Object 0 ICP 15 Handle
Rn+3.2	31:16	Object 0 ICP 14 Handle ID
	15:0	Object 0 ICP 14 Handle
Rn+3.1	31:16	Object 0 ICP 13 Handle ID
	15:0	Object 0 ICP 13 Handle
Rn+3.0	31:16	Object 0 ICP 12 Handle ID
	15:0	Object 0 ICP 12 Handle
Varies (optional)	31:0	Constant Data (optional): Some amount of constant data (possibly 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_GS command (taking the buffer enables into account). The Constant Data arrives in a non-interleaved format.

GRF DWord	Bits	Description
Varies	31:0	<p>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.</p> <p>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.</p> <p>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.</p>

SIMD8 Thread Payload

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

Refer to the 3D Pipeline Stage Overview section for details on those fields that are common among the various pipeline stages.

GRF DWord	Bits	Description
R0.7	31:0	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 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 a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: <div>Format</div> <div>U9</div> Range: <div>Range</div> <div>0-503</div>
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 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]

GRF DWord	Bits	Description
R0.2	31:24	Reserved.
	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 <i>3D Pipeline</i> .
	15:13	Reserved.
	12:0	Reserved.
R0.1-R0.0	31:0	Reserved.
R1.7	31:0	GS Instance ID / URB Return Handle for Object 7 (See R1.0)
R1.6	31:0	GS Instance ID / URB Return Handle for Object 6 (See R1.0)
R1.5	31:0	GS Instance ID / URB Return Handle for Object 5 (See R1.0)
R1.4	31:0	GS Instance ID / URB Return Handle for Object 4 (See R1.0)
R1.3	31:0	GS Instance ID / URB Return Handle for Object 3 (See R1.0)
R1.2	31:0	GS Instance ID / URB Return Handle for Object 2 (See R1.0)
R1.1	31:0	GS Instance ID / URB Return Handle for Object 1 (See R1.0)
R1.0	31:27	GS Instance ID 0. For each input object, the GS unit can spawn multiple threads (instances). This field starts at zero for the first instance of an object and increments for subsequent instances. Format: U5
	26:16	Reserved.
	15:0	URB Return Handle 0. This field is the URB offset where the EU lower channels (DWords 3:0) results are 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

GRF DWord	Bits	Description
R2.4	31:0	Primitive ID 4. This field contains the Primitive ID associated with input object 4 (or the single input object if InstanceCount > 1). Format: U32
R2.3	31:0	Primitive ID 3. This field contains the Primitive ID associated with input object 3 (or the single input object if InstanceCount > 1). Format: U32
R2.2	31:0	Primitive ID 2. This field contains the Primitive ID associated with input object 2 (or the single input object if InstanceCount > 1). Format: U32
R2.1	31:0	Primitive ID 1. This field contains the Primitive ID associated with input object 1 (or the single input object if InstanceCount > 1). Format: U32
R2.0	31:0	Primitive ID 0. This field contains the Primitive ID associated with input object 0 (or the single input object if InstanceCount > 1). Format: U32
The following registers are included only if Include Vertex Handles is enabled and InstanceCount == 1 .		
Rn.7	31:16	Object 7 ICP 0 Handle ID.
	15:0	Object 7 ICP 0 Handle.
Rn.6	31:16	Object 6 ICP 0 Handle ID.
	15:0	Object 6 ICP 0 Handle.
Rn.5	31:16	Object 5 ICP 0 Handle ID.
	15:0	Object 5 ICP 0 Handle.
Rn.4	31:16	Object 4 ICP 0 Handle ID.
	15:0	Object 4 ICP 0 Handle.
Rn.3	31:16	Object 3 ICP 0 Handle ID.
	15:0	Object 3 ICP 0 Handle.
Rn.2	31:16	Object 2 ICP 0 Handle ID.
	15:0	Object 2 ICP 0 Handle.
Rn.1	31:16	Object 1 ICP 0 Handle ID.
	15:0	Object 1 ICP 1 Handle.
Rn.0	31:16	Object 0 ICP 0 Handle ID.
	15:0	Object 0 ICP 0 Handle.
[Rn+1]	255:0	ICP 1 Handle/HandleID for Objects 0-7.
[Rn+2]	255:0	ICP 2 Handle/HandleID for Objects 0-7.
[Rn+3]	255:0	ICP 3 Handle/HandleID for Objects 0-7.
[Rn+4]	255:0	ICP 4 Handle/HandleID for Objects 0-7.

GRF DWord	Bits	Description
[Rn+5]	255:0	ICP 5 Handle/HandleID 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	Reserved.
Rn.6	31:16	Reserved.
	15:0	Reserved.
Rn.5	31:16	ICP 5 Handle ID (if required).
	15:0	ICP 5 Handle (if required).
Rn.4	31:16	ICP 4 Handle ID (if required).
	15:0	ICP 4 Handle (if required).
Rn.3	31:16	ICP 3 Handle ID (if required).
	15:0	ICP 3 Handle (if required).
Rn.2	31:16	ICP 2 Handle ID (if required).
	15:0	ICP 2 Handle (if required).
Rn.1	31:16	ICP 1 Handle ID (if required).
	15:0	ICP 1 Handle (if required).
Rn.0	31:16	ICP 0 Handle ID (if required).
	15:0	ICP 0 Handle.
Varies optional	31:0	Constant Data (optional): Some amount of constant data (possibly 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_GS command (taking the buffer enables into account). The Constant Data arrives in a non-interleaved format.
Varies		Pushed Vertex Data (InstanceCount == 1 Case): 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

GRF DWord	Bits	Description
Rv.4	31:0	Object 4 Vertex 0 Element 0 Component 0
Rv.3	31:0	Object 3 Vertex 0 Element 0 Component 0
Rv.2	31:0	Object 2 Vertex 0 Element 0 Component 0
Rv.1	31:0	Object 1 Vertex 0 Element 0 Component 0
Rv.0	31:0	Object 0 Vertex 0 Element 0 Component 0
Rv+1	31:0	Object 0-7 Vertex 0 Element 0 Component 1
		and so on
Varies		Pushed Vertex Data (InstanceCount > 1 Case): 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.

Thread Request Generation

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

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

Thread Control Information

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

State Variables Included in Thread Control Information

State Variable	Usage	FFs
Kernel Start Pointer	This field, together with the General State Pointer , specifies the starting location (1 st 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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} with m > 1). See CR0 description in <i>ISA Execution Environment</i> .	All FFs spawning threads
Thread Dispatch Priority	The Thread Dispatcher will give priority to those thread requests with Thread Dispatch Priority of HIGH_PRIORITY over those marked as LOW_PRIORITY. Within these two classes of thread requests, the Thread Dispatcher applies a priority order (e.g., round-robin --- though this algorithm is considered a device implementation-dependent detail).	All FFs spawning threads
Floating Point Mode	This determines the initial value of the Floating Point Mode bit of the EU's CR0 architectural register that controls floating point behavior in the EU core. (See ISA.)	All FFs spawning threads
Exceptions Enable	This bitmask controls the exception handing logic in the EU. (See ISA.)	All FFs spawning threads
Sampler Count	This is a <u>hint</u> which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).	All stages supporting sampling (VS, GS, WM)
Binding Table Entry Count	This is a <u>hint</u> which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The notes included in Sampler Count (above) also apply to this field).	All FFs spawning threads

Thread Payload 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	<p>This field identifies this thread within the outstanding threads spawned by the FF unit. This field does <i>not</i> uniquely identify the thread over any significant period.</p> <p>Implementation Note: This field is effectively an "active thread index". It is used on a thread's URB allocation request to identify which thread's handle pool is to source the allocation. It is used upon thread termination to free up the thread's scratch space allocation.</p>	All FFs spawning threads
Binding Table Pointer	<p>This field, together with the Surface State Base Pointer, specifies the starting location of the Binding Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the Surface State Base Pointer.</p> <p>See <i>Shared Functions</i> for a description of a Binding Table.</p>	All FFs spawning threads
Sampler State Pointer	<p>This field, together with the General State Base Pointer, specifies the starting location of the Sampler State Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the General State Base Pointer.</p> <p>See <i>Shared Functions</i> for a description of a Sampler State Table.</p>	All FFs spawning threads which sample (VS, GS, WM)

Fixed Payload Header Field (non-FF-specific)	Description	FFs
Per Thread Scratch Space	<p>This field specifies the amount of scratch space allocated to each thread spawned by the FF unit.</p> <p>The driver must allocate enough contiguous scratch space, starting at the Scratch Space Base Pointer, to ensure that the Maximum Number of Threads can each get Per-Thread Scratch Space size without exceeding the driver-allocated scratch space.</p>	All FFs spawning threads
Handle ID <n>	<p>This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread.</p> <p>Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.</p>	VS, GS, CLIP, SF
URB Return Handle <n>	<p>This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry.</p> <p>Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.</p>	VS, GS, CLIP, SF
Primitive Topology Type	<p>As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (for example, for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread.</p> <p>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.</p> <p>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).</p>	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	<p>This SV identifies the starting GRF register receiving payload URB data.</p> <p>Software is responsible for ensuring that URB data does not overwrite the Fixed or Extended Header portions of the payload.</p>	FFs spawning threads
Vertex URB Entry Read Offset	<p>This SV specifies the starting offset within VUEs from which vertex data is to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into any and all VUEs passed in the payload.</p> <p>This SV can be used to skip over leading data in VUEs that is not required by the stage's threads (e.g., skipping over the Vertex Header data at the SF stage, as that information is not required for setup calculations). Skipping over irrelevant data can only help to improve performance.</p> <p>Specifying a vertex data source extending beyond the end of a vertex entry is UNDEFINED.</p>	VS, GS
Vertex URB Entry Read Length	<p>This SV determines the amount of vertex data (starting at Vertex URB Entry Read Offset) to be read from each VUEs and passed into the payload URB data. It is specified in 256-bit units.</p> <p>A zero value is INVALID (at very least one 256-bit unit must be read).</p> <p>Specifying a vertex data source extending beyond the end of a VUE is UNDEFINED.</p>	

Programming Restrictions: (others may already been mentioned)

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

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.

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	<p>If clear, the SO stage will not forward any topologies down the pipeline.</p> <p>If set, the SO stage will forward topologies associated with Render Stream Select down the pipeline.</p> <p>This bit is used even if SO Function Enable is DISABLED.</p>	<pre> = (3DSTATE_STREAMOUT::Force_Rendering == Force_On) ((3DSTATE_STREAMOUT::Force_Rendering != Force_Off) && !(3DSTATE_GS::Enable && 3DSTATE_GS::Output_Verbose_Size == 0) && !3DSTATE_STREAMOUT::API_Render_Disable && (3DSTATE_DEPTH_STENCIL_STATE::Stencil_TestEnable 3DSTATE_DEPTH_STENCIL_STATE::Depth_TestEnable 3DSTATE_DEPTH_STENCIL_STATE::Depth_WriteEnable 3DSTATE_PS_EXTRA::PS_Valid 3DSTATE_WM::Legacy_Depth_Buffer_Clear 3DSTATE_WM::Legacy_Depth_Buffer_Resolve_Enable 3DSTATE_WM::Legacy Hierarchical_Depth_Buffer_Resolve_Enable)) </pre>

DW1[21]	DW1[20]	Stream Offset	Action
Full legacy mode. HW doesn't LOAD or STORE, it simply updates the MMIO register during stream out. SW can 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)	
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.

PrimTopologyType	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
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	Same as TRILIST	Handled same as TRILIST.
TRILIST_ADJ (N is multiple of 6)	[0] = (0,2,4); [1] = (6,8,10); ...; [(N/6)-1] = (N-6,N-4,N-2)	
TRISTRIP (N >= 3) REORDER_LEADING	[0] = (0,1,2); [1] = (1,3,2); [k even] = (k,k+1,k+2) [k odd] = (k,k+2,k+1) [N-3] = (see above)	"Odd" triangles have vertices reordered to yield increasing leading vertices starting with v0.
TRISTRIP (N >= 3) REORDER_TRAILING	[0] = (0,1,2); [1] = (2,1,3); [k even] = (k,k+1,k+2) [k odd] = (k+1,k,k+2) [N-3] = (see above)	"Odd" triangles have vertices reordered to yield increasing trailing vertices starting with v2.
TRISTRIP_REV (N >= 3) REORDER_LEADING	[0] = (0,2,1) [1] = (1,2,3);...; [k even] = (k,k+2,k+1) [k odd] = (k,k+1,k+2) [N-3] = (see above)	"Even" triangles have vertices reordered to yield increasing leading vertices starting with v0.
TRISTRIP_REV (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.

PrimTopologyType	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
TRISTRIP_ADJ (N even, N >= 6) REORDER_LEADING	N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (2,6,4); ...; N > 10: [0] = (0,2,4); [1] = (2,6,4); ...; [k>1, even] = (2k, 2k+2, 2k+4); [k>2, odd] = (2k, 2k+4, 2k+2);...; Trailing object: [(N/2)-3, even] = (N-6,N-4,N-2); [(N/2)-3, odd] = (N-6,N-2,N-4);	"Odd" objects have vertices reordered to yield increasing-by-2 leading vertices starting with v0.
TRISTRIP_ADJ (N even, N >= 6) REORDER_TRAILING	N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (4,2,6); ...; N > 10: [0] = (0,2,4); [1] = (4,2,6); ...; [k>1, even] = (2k, 2k+2, 2k+4); [k>2, odd] = (2k+2,2k, 2k+4);...; Trailing object: [(N/2)-3, even] = (N-6,N-4,N-2); [(N/2)-3, odd] = (N-4,N-6,N-2);	"Odd" objects have vertices reordered to yield increasing-by-2 trailing vertices starting with v4.
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); ...; [N-3] = (0, N-2, N-1);	
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.

PrimTopologyType	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
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 be set, else it must be cleared).

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

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

Project	Surface Format Name	Security
BDW	R32G32B32A32_FLOAT	
BDW	R32G32B32A32_SINT	
BDW	R32G32B32A32_UINT	
BDW	R32G32B32_FLOAT	
BDW	R32G32B32_SINT	
BDW	R32G32B32_UINT	
BDW	R32G32_FLOAT	
BDW	R32G32_SINT	
BDW	R32G32_UINT	
BDW	R32_SINT	
BDW	R32_UINT	
BDW	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

For BDW, 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

Signal	Description	Formula
CLIP_INT::Viewport Z ClipTest Enable	This field is used to control whether the Viewport Z extents (near, far) are considered in VertexClipTest.	= 3DSTATE_RASTER::Viewport Z ClipTest Enable
CLIP_INT::User Clip Distance Cull Test Enable Bitmask	<p>This 8-bit mask field selects which of the 8 user clip distances against which trivial reject/trivial accept determination needs to be made (does not cause a must clip).</p> <p>DX10 allows simultaneous use of ClipDistance and Cull Distance test of up to 8 distances.</p>	<p>= (3DSTATE_CLIP::ForceUser Clip Distance Cull Test Enable Bitmask == Force) ? 3DSTATE_CLIP::User Clip Distance Cull Test Enable Bitmask :</p> <p>3DSTATE_GS::GS_Enable ? 3DSTATE_GS::GS User Clip Distance Cull Test Enable Bitmask :</p> <p>3DSTATE_DS::DS_Enable ? 3DSTATE_DS::DS User Clip Distance Cull Test Enable Bitmask :</p> <p>3DSTATE_INT::VS_Enable ? 3DSTATE_VS::VS User Clip Distance Cull Test Enable Bitmask :</p> <p>0</p>
CLIP_INT::User Clip Distance Clip Test Enable Bitmask	<p>This 8-bit mask field selects which of the 8 user clip distances against which trivial reject/trivial accept determination needs to be made (does not cause a must clip).</p> <p>DX10 allows simultaneous use of ClipDistance and Clip Distance test of up to 8 distances.</p>	<p>= (3DSTATE_CLIP::ForceUser Clip Distance Clip Test Enable Bitmask == Force) ? 3DSTATE_CLIP::User Clip Distance Clip Test Enable Bitmask :</p> <p>3DSTATE_GS::GS_Enable ? 3DSTATE_GS::GS User Clip Distance Clip Test Enable Bitmask :</p> <p>3DSTATE_DS::DS_Enable ? 3DSTATE_DS::DS User Clip Distance Clip Test Enable Bitmask :</p> <p>3DSTATE_VS::VS_Enable ? 3DSTATE_VS::VS User Clip Distance Clip Test Enable Bitmask :</p> <p>0</p>

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.

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

The following algorithm is used by VertexClipTest:

```
//
// Vertex ClipTest
//
// On input:
// if (CLIP.PreMapped)
//   x,y are viewport mapped
//   z is NDC ([0,1] is visible)
// else
//   x,y,z are NDC (post-perspective divide)
//   w is always 1/w
//
// Initialize outCodes to "inside"
//
outCode[*] = 0
//
// Check if w is NaN
// Any object containing one of these "bad" vertices will likely be discarded
//
if (ISNAN(homogeneous x,y,z,w or enabled ClipDistance value)
{
    outCode[BAD] = 1
}
//
// If 1/w is negative, w is negative and therefore outside of the w=0 plane
//
//
rhw_neg = ISNEG(rhw)
if (rhw_neg)
{
    outCode[NEGW] = 1
}
//
// View Volume Clip Test
// If Premapped, the 2D viewport is defined in screen space
// otherwise the canonical NDC viewvolume applies ([-1,1])
//
if (CLIP_STATE.PreMapped)
{
    vp_XMIN = CLIP_STATE.VP_XMIN
    vp_XMAX = CLIP_STATE.VP_XMAX
    vp_YMIN = CLIP_STATE.VP_YMIN
    vp_YMAX = CLIP_STATE.VP_YMAX
} else {
    vp_XMIN = -1.0f
    vp_XMAX = +1.0f
    vp_YMIN = -1.0f
    vp_YMAX = +1.0f
}

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

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

```

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

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

#ifdef (BW-E0)
    if (rhw_neg) {
        outCode[VP_ZMIN] = (z >= vp_ZMIN)
        outCode[VP_ZMAX] = (z <= vp_ZMAX)
    }
#endif
    if (rhw_neg) {
        outCode[VP_ZMIN] = (z > vp_ZMIN)
        outCode[VP_ZMAX] = (z < vp_ZMAX)
    }
}
//
// Guardband Clip Test
//
if (CLIP_STATE.GuardbandClipTestEnable) {
    gb_XMIN = CLIP_STATE.Viewport[vpindex].GB_XMIN
    gb_XMAX = CLIP_STATE.Viewport[vpindex].GB_XMAX
    gb_YMIN = CLIP_STATE.Viewport[vpindex].GB_YMIN
    gb_YMAX = CLIP_STATE.Viewport[vpindex].GB_YMAX
    outCode[GB_XMIN] = (x < gb_XMIN)
    outCode[GB_XMAX] = (x > gb_XMAX)
    outCode[GB_YMIN] = (y < gb_YMIN)
    outCode[GB_YMAX] = (y > gb_YMAX)

#ifdef (BW-E0)
    if (rhw_neg) {
        outCode[GB_XMIN] = (x >= gb_XMIN)
        outCode[GB_XMAX] = (x <= gb_XMAX)
        outCode[GB_YMIN] = (y >= gb_YMIN)
        outCode[GB_YMAX] = (y <= gb_YMAX)
    }
#endif
    if (rhw_neg) {
        outCode[GB_XMIN] = (x > gb_XMIN)
        outCode[GB_XMAX] = (x < gb_XMAX)
        outCode[GB_YMIN] = (y > gb_YMIN)
        outCode[GB_YMAX] = (y < gb_YMAX)
    }
}
//
// Handle case where either VP or GB disabled (but not both)
// In this case, the disabled set take on the outcodes of the enabled set

```

```
//
if (CLIP_STATE.ViewportXYClipTestEnable && !CLIP_STATE.GuardbandClipTestEnable) {
    outCode[GB_XMIN] = outCode[VP_XMIN]
    outCode[GB_XMAX] = outCode[VP_XMAX]
    outCode[GB_YMIN] = outCode[VP_YMIN]
    outCode[GB_YMAX] = outCode[VP_YMAX]
} else if (!CLIP_STATE.ViewportXYClipTestEnable && CLIP_STATE.GuardbandClipTestEnable) {
    outCode[VP_XMIN] = outCode[GB_XMIN]
    outCode[VP_XMAX] = outCode[GB_XMAX]
    outCode[VP_YMIN] = outCode[GB_YMIN]
    outCode[VP_YMAX] = outCode[GB_YMAX]
}
//
// X/Y/Z NaN Handling
//
xyorgben = (CLIP_STATE.ViewportXYClipTestEnable || CLIP_STATE.GuardbandClipTestEnable)
if (isNaN(x)) {
    outCode[GB_XMIN] = xyorgben
    outCode[GB_XMAX] = xyorgben
    outCode[VP_XMIN] = xyorgben
    outCode[VP_XMAX] = xyorgben
}
if (isNaN(y)) {
    outCode[GB_YMIN] = xyorgben
    outCode[GB_YMAX] = xyorgben
    outCode[VP_YMIN] = xyorgben
    outCode[VP_YMAX] = xyorgben
}
if (isNaN(z)) {
    outCode[VP_ZMIN] = CLIP_STATE.ViewportZClipTestEnable
    outCode[VP_ZMAX] = CLIP_STATE.ViewportZClipTestEnable
}
//
// UserClipFlags
//
ExamineUCFs
for (i=0; i<7; i++)
{
    outCode[UC0+i] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[i]
}
outCode[UC7] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
```


Object Staging

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

Partial Object Removal

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

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

ClipDetermination Function

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

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

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

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

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way, objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

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

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

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

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

//
// Trivial Accept
//
// For an object to be TA, it must be TA against the VPZ and GB, not TR,
// and considered TA against the UC planes and NEGW
// If the UCMC mode is disabled, an object is considered TA against the UC
// as long as it isn't TR against the UC.
// If the UCMC mode is enabled, then the object really has to be TA against the UC
// to be considered TA
// In this way, enabling the UCMC mode will force clipping if the object is neither
// TA or TR against the UC
//
TA = !TR && TA_GB && TA_VPZ && TA_NEGW
```

```
UCMC = CLIP_STATE.UserClipFlagsMustClipEnable
TA    = TA && ( (UCMC && TA_UC) || (!UCMC && !TR_UC) )

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

ClipMode State

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

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

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

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

NORMAL ClipMode

In NORMAL mode, objects will be discarded if TR or BAD, passed through if TA, and passed to a CLIP thread if MC. 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 and all other (TA, MC) objects are passed to CLIP threads. Usage of this mode assumes that the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are sufficient at least in respect to determining trivial reject.

REJECT_ALL ClipMode

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

ACCEPT_ALL ClipMode

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

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

Object Pass-Through

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

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

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
POINTLIST	POINTLIST	
POINTLIST_BF	POINTLIST_BF	
LINELIST	LINELIST	
LINELIST_ADJ	LINELIST	Adjacent vertices removed.
LINESTRIP	LINESTRIP	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT	LINESTRIP	Adjacent vertices removed. LINESTRIP_ADJ_CONT is added 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.
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

Introduction for this section under development.

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. <i>Primitive Assembly</i> for a list of primitive types supported by the SF unit. See <i>3D Pipeline</i> for descriptions of these topologies. Notes: The CLIP unit will convert any primitive with adjacency (3DPRIMxxx_ADJ) it receives from the pipeline into the corresponding primitive without adjacency (3DPRIMxxx). QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit. Software must use a GS thread to convert these to some other (supported) primitive type. If an object is clipped by the hardware clipper, the CLunit would force this field to “3DPRIM_POLYGON”. SFunit would process this incoming object just as it would any other “3DPRIM_POLYGON”. SFunit selects vertex 0 as the provoking vertex.
primStart,primEnd	boolean	Indicate vertex's position within the primitive topology
vInX[]	float	Vertex X position (screen space or NDC space)
vInY[]	float	Vertex Y position (screen space or NDC space)
vInZ[]	float	Vertex Z position (screen space or NDC space)
vInInvW[]	float	Reciprocal of Vertex homogeneous (clip space) W
hVUE[]	URB address	Points to the vertex's data stored in the URB (one VUE handle per vertex)
renderTargetArrayIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in subsequent processing.

Variable	Type	Description
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	>=3
3DPRIM_TRISTRIP_REVERSE	
3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	>=3
3DPRIM_LINELIST	
3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	>=2
3DPRIM_RECTLIST	
3DPRIM_POINTLIST	nonzero
3DPRIM_POINTLIST_BF	

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. <i>Primitive Assembly</i>
nV	uint	The number of object vertices passed to Object Setup. <i>Primitive Assembly</i>
v[0..nV-1]*	various	Data arrays associated with <u>object</u> vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. <i>Primitive Assembly</i>
invertOrientation	enum	Indicates whether the orientation (CW or CCW winding order) of the vertices of a triangle object should be inverted. Ignored for non-triangle objects.
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 <i>v[]</i> 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 VPIIndex/RTAIndex applies to the objects within the topology.

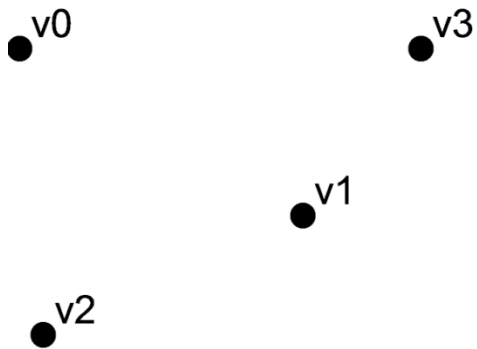
VPIndex/RTAIndex Selection

PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V2.VPIndex → Line(V2,V3) ...
LINESTRIP LINESTRIP_BF LINESTRIP_CONT LINESTRIP_CONT_BF	The leading vertex of each line segment supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V1.VPIndex → Line(V1,V2) ... NOTE: If the VPIndex changes within the topology, shared vertices will be processed (mapped) multiple times.
TRILIST RECTLIST	The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects. V0.VPIndex → Tri(V0,V1,V2) V3.VPIndex → Tri(V3,V4,V5) ...
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.

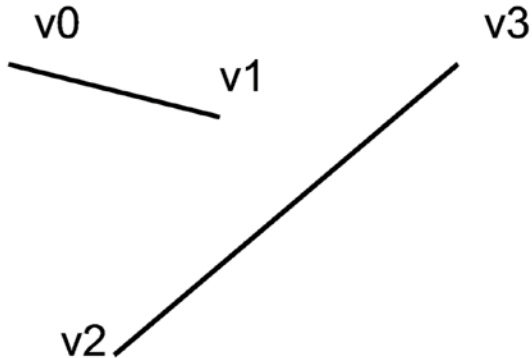
```
PointListDecomposition()
{
    objectType = 3DOBJ_POINT
    nV = 1
    provokingVtx = 0
    if (primType == 3DPRIM_POINTLIST)
    {
        backFacing = FALSE
    }
    else // primType == 3DPRIM_POINTLIST_BACKFACING
    {
        backFacing = TRUE
    }

    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.

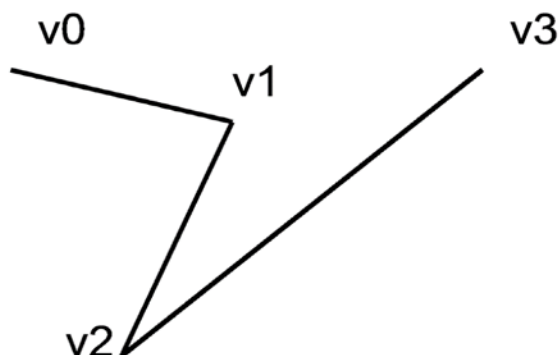
```

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
    }
}

```

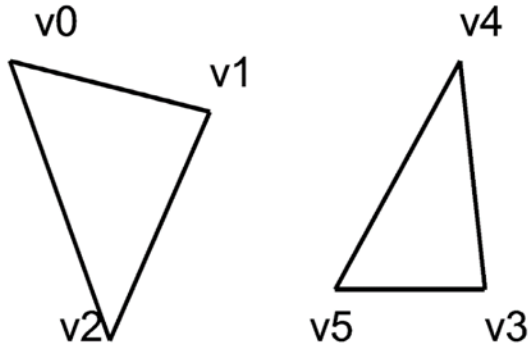


```
    for each (vertex I in [0..vertexCount-1])  
    {  
        v[0] arrays ← vIn[i] arrays  
        v[1] arrays ← vIn[i+1] arrays  
        ObjectSetup()  
        continueStipple = TRUE  
    }  
}
```

Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

3DPRIM_TRILIST Primitive



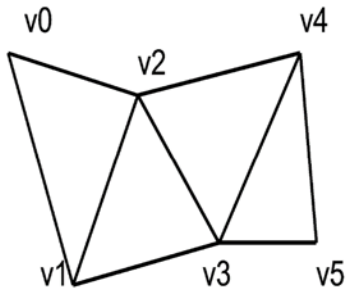
The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is 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
    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 trisrip primitive commands (for example, due to ring or batch buffer size restrictions), two trisrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

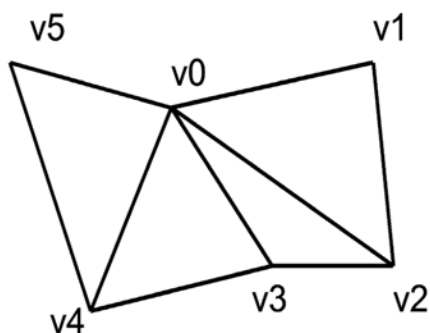
```
TriangleStripDecomposition()
{
    objectType = 3DOBJ_TRIANGLE
    nV = 3
    provokingVtx = Triangle List/Strip Provoking Vertex Select
    if (primType == 3DPRIM_TRISTRIP)
        invertOrientation = FALSE
    else // primType == 3DPRIM_TRISTRIP_REVERSE
        invertOrientation = TRUE
    polyStippleEnable = TRUE
    for each (vertex I in [0..vertexCount-3])
    {
        v[0] arrays ← vIn[i] arrays
        v[1] arrays ← vIn[i+1] arrays
        v[2] arrays ← vIn[i+2] arrays
        ObjectSetup()
        invertOrientation = ! invertOrientation
    }
}
```

```
}  
}
```

Triangle Fan Decomposition

The 3DPRIM_TRIFAN and 3DPRIM_TRIFAN_NOSTIPPLE primitives specify a series of triangles arranged in a fan, as illustrated below.

3DPRIM_TRIFAN Primitive



The decomposition process divides the fan into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v0,v2,v3; v0,v3,v4; and so on. As there is no alternation in the vertex winding order, the *invertOrientation* variable is output as FALSE unconditionally. The *provokingVertex* of each object is taken from the **Triangle Fan Provoking Vertex** state variable, as programmed via SF_STATE.

Primitives of type 3DPRIM_TRIFAN_NOSTIPPLE are decomposed exactly the same way, except the *polyStippleEnable* variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```
TriangleFanDecomposition()  
{  
    objectType = 3DOBJ_TRIANGLE  
    nV = 3  
    invertOrientation = FALSE  
    provokingVtx = Triangle Fan Provoking Vertex Select  
    if (primType == 3DPRIM_TRIFAN)  
        polyStippleEnable = TRUE  
    else // primType == 3DPRIM_TRIFAN_NOSTIPPLE  
        polyStippleEnable = FALSE  
  
    v[0] arrays ← vIn[0] arrays  
    // the 1st vertex is common  
  
    for each (vertex I in [1..vertexCount-2])  
    {  
        v[1] arrays ← vIn[i] arrays  
        v[2] arrays ← vIn[i+1] arrays  
        ObjectSetup()  
    }  
}
```

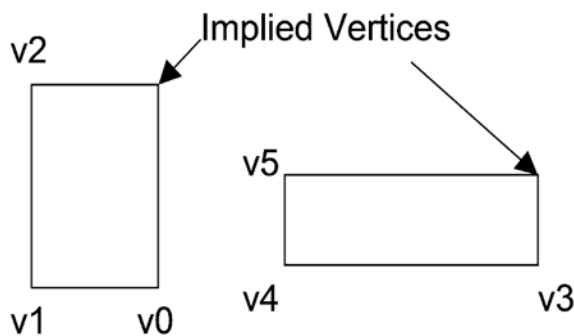
Polygon Decomposition

The 3DPRIM_POLYGON primitive is identical to the 3DPRIM_TRIFAN primitive with the exception that the *provokingVtx* is overridden with 0. This support has been added specifically for OpenGL support, avoiding the need for the driver to change the provoking vertex selection when switching between trifan and polygon primitives.

Rectangle List Decomposition

The 3DPRIM_RECTLIST primitive command specifies a list of independent, axis-aligned rectangles. Only the lower right, lower left, and upper left vertices (in that order) are included in the command – the upper right vertex is derived from the other vertices (in Object Setup).

3DPRIM_RECTLIST Primitive



The decomposition of the 3DPRIM_RECTLIST primitive is identical to the 3DPRIM_TRILIST decomposition, with the exception of the *objectType* variable.

```
RectangleListDecomposition()
{
    objectType = 3DOBJ_RECTANGLE
    nV = 3
    invertOrientation = FALSE
    provokingVtx = 0
    for each (vertex I in [0..vertexCount-3] by 3)
    {
        v[0] arrays ← vIn[i] arrays
        v[1] arrays ← vIn[i+1] arrays
        v[2] arrays ← vIn[i+2] arrays
        ObjectSetup()
    }
}
```

Object Setup

The Object Setup subfunction of the SF stage takes the post-viewport-transform data associated with each vertex of a basic object and computes various parameters required for scan conversion. This includes generation of implied vertices, translations and adjustments on vertex positions, and culling (removal) of certain classes of objects. The final object information is passed to the Windower/Masker (WM) stage where the object is rasterized into pixels.

Invalid Position Culling (Pre/Post-Transform)

At input the the SF stage, any objects containing a floating-point NaN value for Position X, Y, Z, or RHW will be unconditionally discarded. Note that this occurs on an object (not primitive) basis.

If Viewport Transformation is enabled, any objects containing a floating-point NaN value for post-transform Position X, Y or Z will be unconditionally discarded.

Viewport Transformation

If the **Viewport Transform Enable** bit of SF_STATE is ENABLED, a viewport transformation is applied to each vertex of the object.

The VPIndex associated with the leading vertex of the object is used to obtain the **Viewport Matrix Element** data from the corresponding element of the SF_VIEWPORT structure in memory. For each object vertex, the following scale and translate transformation is applied to the position coordinates:

$$x' = m00 * x + m30$$

$$y' = m11 * y + m31$$

$$z' = m22 * z + m32$$

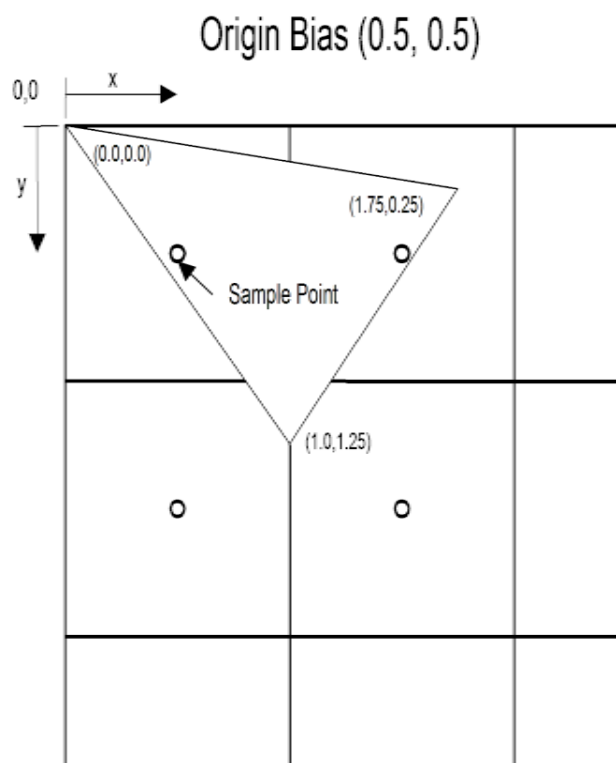
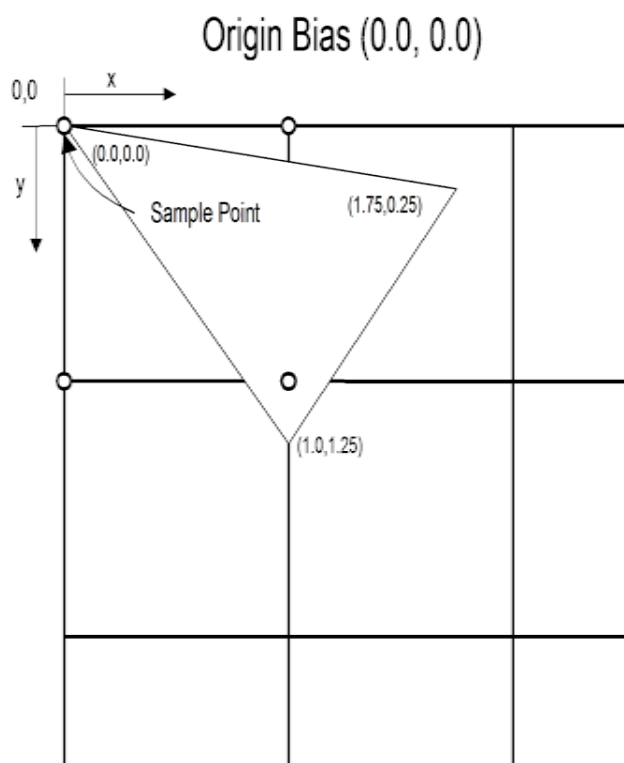
Software is responsible for computing the matrix elements from the viewport information provided to it from the API.

Destination Origin Bias

The positioning of the pixel sampling grid is programmable and is controlled by the **Destination Origin Horizontal/Vertical Bias** state variables (set via SF_STATE). If these bias values are both 0, pixels are sampled on an integer grid. Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0,0) falls within the primitive.

If the bias values are both 0.5, pixels are sampled on a "half" integer grid (i.e., X.5, Y.5). Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0.5,0.5) falls within the primitive. This positioning of the sample grid corresponds with the OpenGL rasterization rules, where "fragment centers" lay on a half-integer grid. It also corresponds with the Intel740 rasterizer (though that device did not employ "top left" rules).

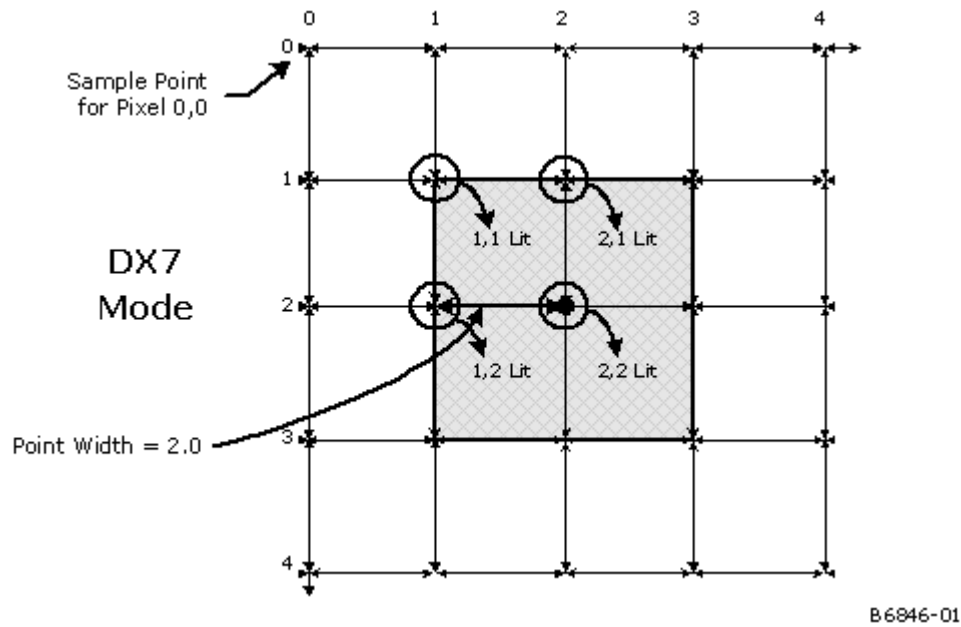
Note that subsequent descriptions of rasterization rules for the various objects will be with reference to the pixel sampling grid.

Destination Origin Bias

Point Rasterization Rule Adjustment

POINT objects are rasterized as square RECTANGLEs, with one exception: The **Point Rasterization Rule** state variable (in SF_STATE) controls the rendering of point object edges that fall directly on pixel sample points, as the treatment of these edge pixels varies between APIs.

RASTRULE_UPPER_LEFT

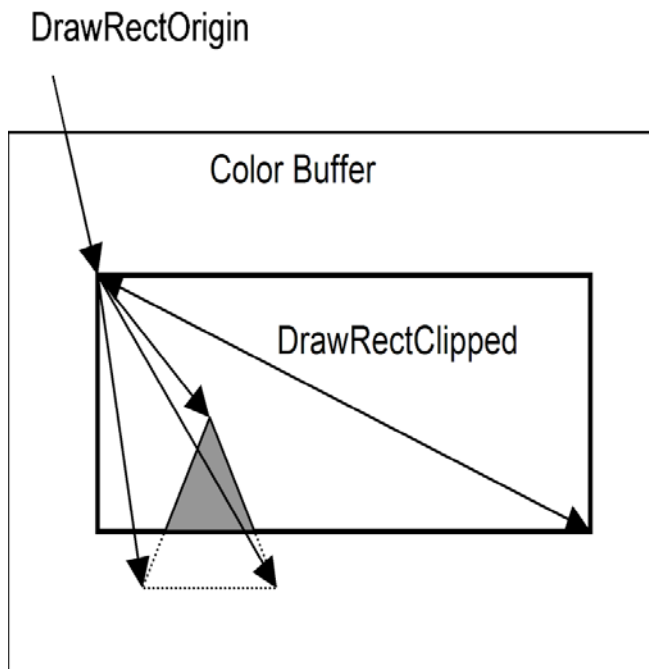


Drawing Rectangle Offset Application

The Drawing Rectangle Offset subfunction offsets the object's vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window's or viewport's upper-left corner.

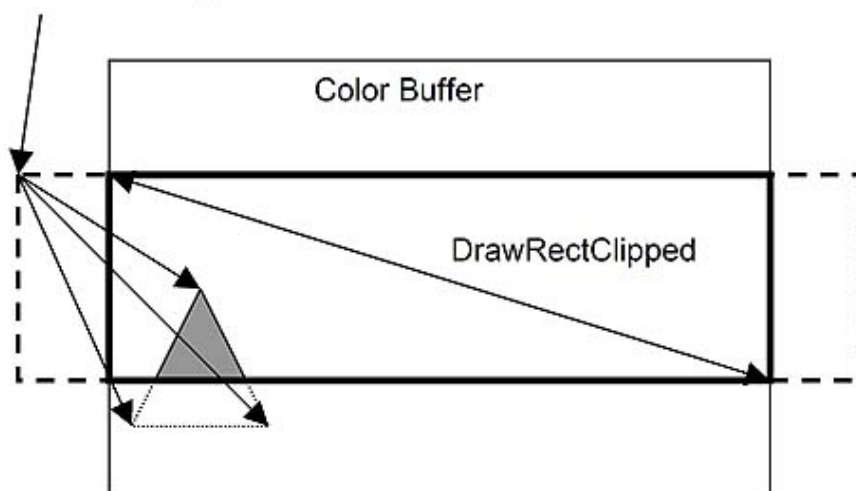
Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the "clipped" draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.

Onscreen Draw Rectangle



Partially-offscreen Draw Rectangle

DrawRectOrigin



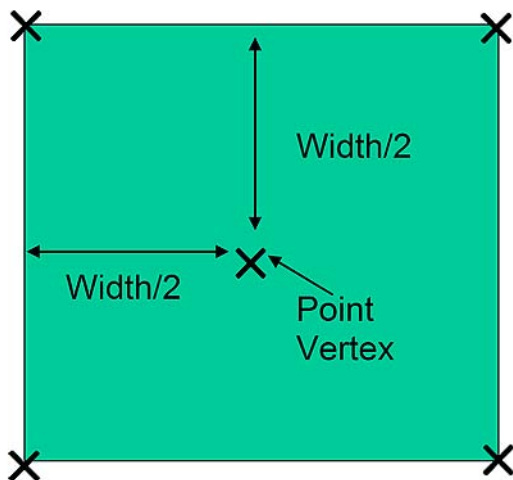
3DSTATE_DRAWING_RECTANGLE

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

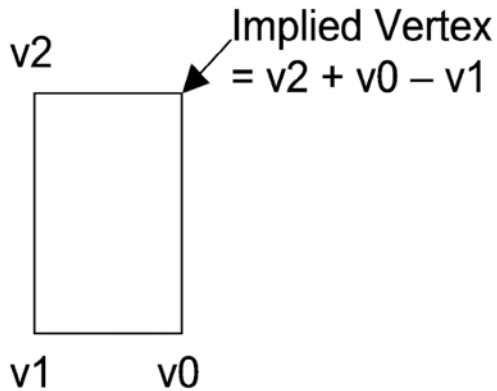


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

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

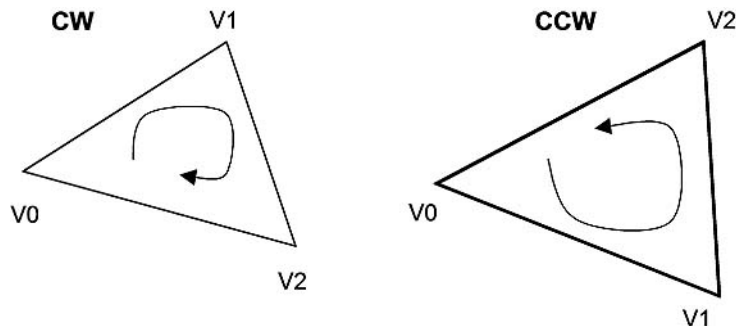
Triangle Orientation (Face) Culling

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 from the ordering of the vertices used to process them.

The **Cull Mode** state variable in SF_STATE specifies how triangles are discarded according to their resultant orientation. See [Degenerate Objects](#).

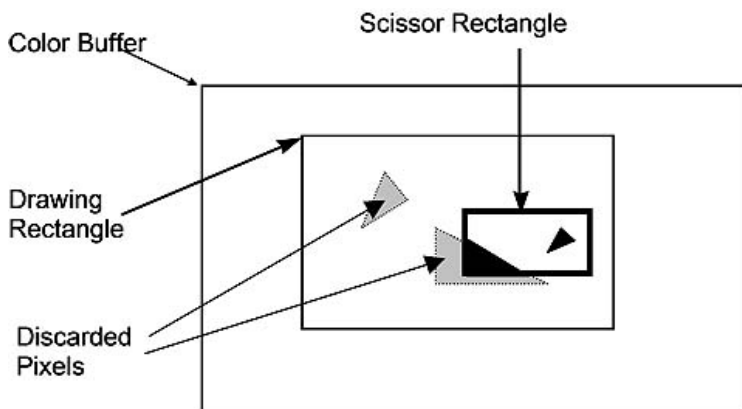
Cull Mode

CullMode	Definition
CULLMODE_NONE	The face culling operation is disabled.
CULLMODE_FRONT	Triangles with “front facing” orientation are discarded.
CULLMODE_BACK	Triangles with “back facing” orientation are discarded.
CULLMODE_BOTH	All triangles are discarded.

Scissor Rectangle Clipping

A *scissor* operation can be used to restrict the extent of rendered pixels to a screen-space aligned rectangle. If the scissor operation is enabled, portions of objects falling outside of the intersection of the scissor rectangle and the clipped draw rectangle are clipped (pixels discarded).

The scissor operation is enabled by the **Scissor Rectangle Enable** state variable in **SF_STATE**. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding **SF_VIEWPORT** structure. Up to 16 structures are supported. The **Scissor Rectangle X,Y Min,Max** fields of the **SF_VIEWPORT** structure defines a scissor rectangle as a rectangle in integer pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. The scissor rectangle is defined relative to the Drawing Rectangle to better support the OpenGL API. (OpenGL specifies the “Scissor Box” in window-relative coordinates). This allows instruction buffers with embedded Scissor Rectangle definitions to remain valid even after the destination window (drawing rectangle) moves.



Specifying either scissor rectangle $xmin > xmax$ 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. 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.

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.

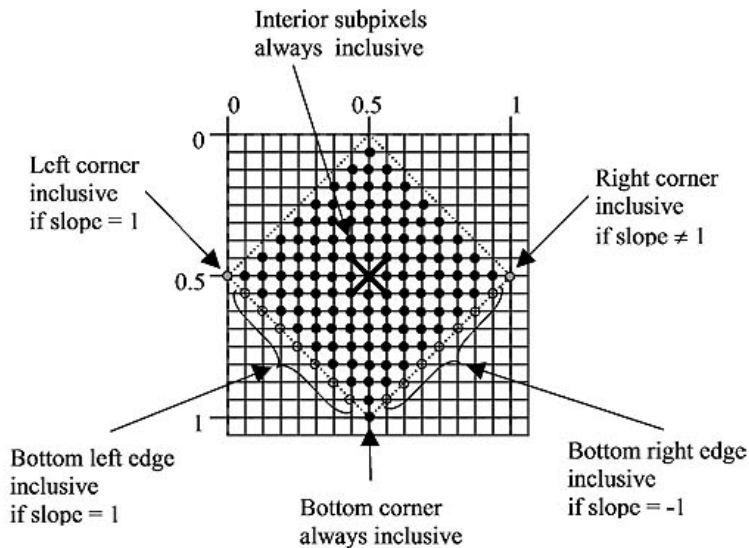
When the **LineWidth** is set to zero, the device will use special rules to rasterize "cosmetic" lines. The rasterization rules also comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

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 $\frac{1}{2}$.

The subpixels falling on the edges of the diamond (Manhattan distance = $\frac{1}{2}$) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range $(-1, 1)$ touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range $(1, +\infty]$ or $[-\infty, -1)$ touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) – where if the right corner was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this “passing through” the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another

(continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would 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” for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```

delta_x          = point.x - sample.x
delta_y          = point.y - sample.y
distance         = abs(delta_x) + abs(delta_y)
interior         = (distance < 0.5)
bottom_corner    = (delta_x == 0.0) && (delta_y == 0.5)
left_corner      = (delta_x == -0.5) && (delta_y == 0.0)
right_corner     = (delta_x == 0.5) && (delta_y == 0.0)
bottom_left_edge = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

```

```

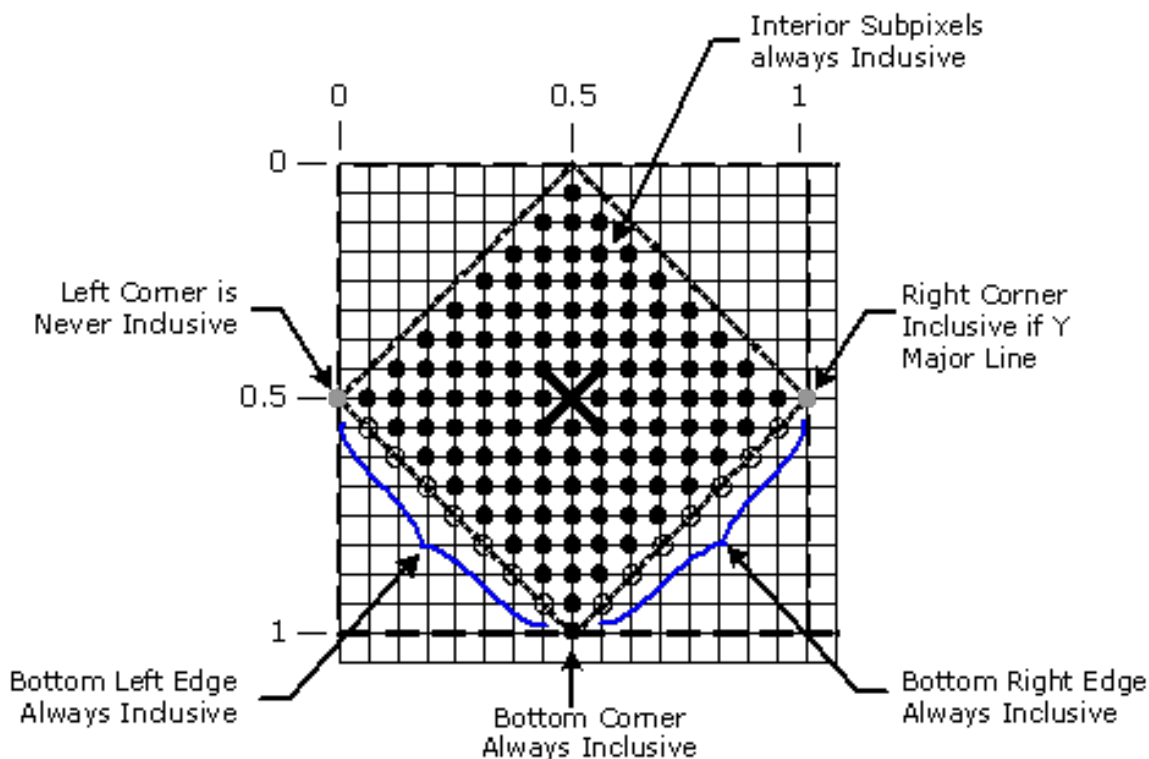
inside = interior || bottom_corner || (slopePosOne ? left_corner : right_corner) ||
(slopePosOne && left_edge) || (slopeNegOne && right_edge)

```

GIQ (Diamond) Sampling Rules – DX10 Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is DISABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



B6849-01

The solid-colored subpixels are considered “interior” to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than $\frac{1}{2}$.

The subpixels falling on the edges of the diamond (Manhattan distance = $\frac{1}{2}$) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range $(-1, 1)$ touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as $-1 \leq \text{slope} \leq 1$).** Including the right corner subpixel ensures that lines with slopes in the range $(>1, +\infty]$ or $[-\infty, <-1)$ touch a diamond even when they cross exactly between pixel diamonds.

3. **The left corner subpixel is never inclusive.** For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
4. **The subpixels along the bottom left edge are always inclusive.** This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would cause the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside", the pixel is only lit when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.
5. **The subpixels along the bottom right edge are always inclusive.** Same as case as (4), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (XMajor).

```

delta_x      = point.x - sample.x
delta_y      = point.y - sample.y
distance     = abs(delta_x) + abs(delta_y)
interior     = (distance < 0.5)
bottom_corner      = (delta_x == 0.0)  && (delta_y == 0.5)
left_corner       = (delta_x == -0.5) && (delta_y == 0.0)
right_corner      = (delta_x == 0.5)  && (delta_y == 0.0)
bottom_left_edge  = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

inside = interior || bottom_corner || (!XMajor && right_corner) || ( bottom_left_edge
|| ( bottom_right_edge)

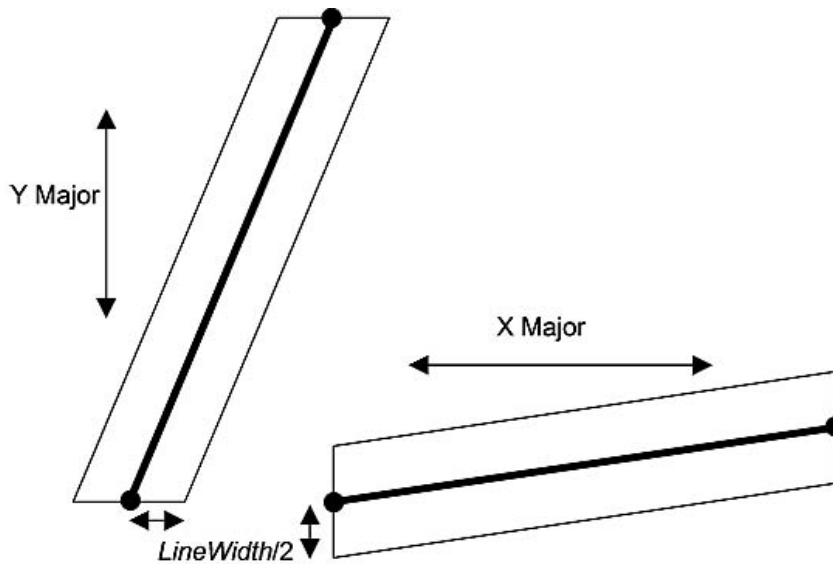
```

Non-Antialiased Wide Line Rasterization

Non-anti-aliased, non-zero-width lines are rendered as parallelograms that are centered on, and aligned to, the line joining the endpoint vertices. Pixels sampled interior to the parallelogram are rendered; pixels sampled exactly on the parallelogram edges are rendered according to the polygon “top left” rules.

The parallelogram is formed by first determining the major axis of the line (diagonal lines are considered x-major). The corners of the parallelogram are computed by translating the line endpoints by $\pm(\text{Line Width} / 2)$ in the direction of the minor axis, as shown in the following diagram.

Non-Antialiased Line Rasterization

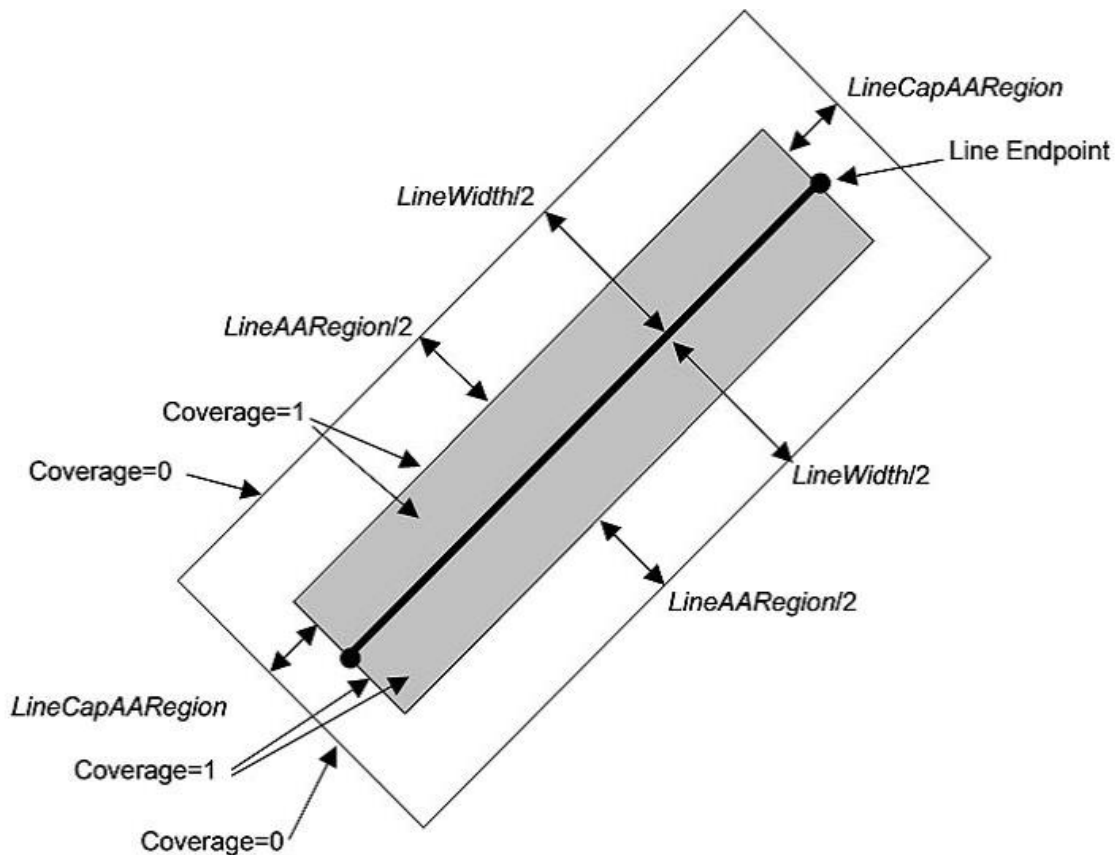


Anti-Aliased Line Rasterization

Anti-aliased lines are rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices. For each pixel in the rectangle, a fractional coverage value (referred to as Antialias Alpha) is computed – this coverage value is normally used to attenuate the pixel’s alpha in the pixel shader thread. The resultant alpha value is therefore available for use in those downstream pixel pipeline stages to generate the desired effect (e.g., use the attenuated alpha value to modulate the pixel’s color, and add the result to the destination color, etc.). Note that software is required to explicitly program the pixel shader and pixel pipeline to obtain the desired anti-aliasing effect – the device simply makes the coverage-attenuated pixel alpha values available for use in the pixel shader.

The dimensions of the rendered rectangle, and the parameters controlling the coverage value computation, are programmed via the **Line Width**, **Line AA Region**, and **Line Cap AA Region** state variables, as shown below. The edges parallel to the line are located at the distance $(\text{LineWidth}/2)$ from the line (measured in screen pixel units perpendicular to the line). The end-cap edges are perpendicular to the line and located at the distance (LineCapAARegion) from the endpoints.

Anti-aliased Line Rasterization



Along the parallel edges, the coverage values ramp from the value 0 at the very edges of the rectangle to the value 1 at the perpendicular distance ($LineAARegion/2$) from a given edge (in the direction of the line). A pixel's coverage value is computed with respect to the closest edge. In the cases where $(LineAARegion/2) < (LineWidth/2)$, this results in a region of fractional coverage values near the edges of the rectangle, and a region of "fully-covered" coverage values (i.e., the value 1) at the interior of the line. When $(LineAARegion/2) == (LineWidth/2)$, only pixel sample points falling exactly on the line can generate fully-covered coverage values. If $(LineAARegion/2) > (LineWidth/2)$, no pixels can be fully-covered (it is expected that this case is not typically desired).

Along the end cap edges, the coverage values ramp from the value 1 at the line endpoint to the value 0 at the cap edge – itself at a perpendicular distance ($LineCapAARegion$) from the endpoint. Note that, unlike the line-parallel edges, there is only a single parameter ($LineCapAARegion$) controlling the extension of the line at the end caps and the associated coverage ramp.

The regions near the corners of the rectangle have coverage values influenced by distances from both the line-parallel and end cap edges – here the two coverage values are multiplied together to provide a composite coverage value.

The computed coverage value for each pixel is passed through the Windower Thread Dispatch payload. The Pixel Shader kernel should be passed (unmodified) by the shader to the Render Cache as part of its output message.

SF Pipeline State Summary

3DSTATE_RASTER

3DSTATE_RASTER

Signal	SF_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table: WM_INT::Multisample Rasterization Mode in 3D Pipeline Windower] > Windower Pipelined State > 3DSTATE_WM > 3DSTATE_WM

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	Enables computation and application of Global Depth Offset for SOLID objects.
Formula	= 3DSTATE_RASTER:: Global Depth Offset Enable Solid

Signal	SF_INT::Global Depth Offset Enable Wireframe
Description	Enables computation and application of Global Depth Offset when triangles are rendered in WIREFRAME mode.
Formula	= 3DSTATE_RASTER:: Global Depth Offset Enable Wireframe

Signal	SF_INT::Global Depth Offset Enable Point
Description	Enables computation and application of Global Depth Offset when triangles are rendered in POINT mode.
Formula	= 3DSTATE_RASTER::Global Depth Offset Enable Point

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 back-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. WYZW.

Each of the first or last set of 16 attributes also has a 5-bit **Source Attribute** field which 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	$(A1-A0)+1$ $(A1-A0) \leq -0.5$ $(A1-A0)-1$ $(A1-A0) \geq 0.5$ $(A1-A0)$ otherwise	$(A2-A0)+1$ $(A2-A0) \leq -0.5$ $(A2-A0)-1$ $(A2-A0) \geq 0.5$ $(A2-A0)$ otherwise

Point Sprites

Normally all vertex attributes (including texture coordinates) other than position are simply replicated from the incoming point center vertex to the generated point object (corner) vertices. However, both DX9 and OGL support “sprite points”, where some/all texture coordinates are replaced with full-scale 2D texture coordinates.

A 32-bit **PointSprite TextureCoordinate Enable** bit mask controls whether the corresponding vertex attribute is to be replaced by a sprite point texture coordinate. The global (not per-attribute) **Point Sprite TextureCoordinate Origin** field controls how the point object vertex (top/bottom, left/right) texture coordinates are generated:

UPPERLEFT	Left	Right
Top	(0,0,0,1)	(1,0,0,1)
Bottom	(0,1,0,1)	(1,1,0,1)

LOWERLEFT	Left	Right
Top	(0,1,0,1)	(1,1,0,1)
Bottom	(0,0,0,1)	(1,0,0,1)

The state used by “setup backend” is defined by the following inline state packet.

3DSTATE_SBE

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.

3DSTATE_WM

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	<pre> = (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))) </pre>
Signal	WM_INT::Pixel Shader Computed Depth Mode
Description	This field specifies the computed depth mode for the pixel shader.
Formula	<pre> = (3DSTATE_PS_EXTRA::ForceComputedDepth == Force) ? 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode : (WM_INT::WM_HZ_OP WM_INT::RT Independent Rasterization Enable) ? PSCDEPTH_OFF: 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode </pre>
Signal	WM_INT::Pixel Shader Uses Source Depth
Description	This bit, if ENABLED, indicates that the PS kernel requires the source depth value (vPos.z) to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source Depth

Signal	WM_INT::Pixel Shader Uses Source W
Description	This bit, if ENABLED, indicates that the PS kernel requires the interpolated source W value (vPos.w) to be passed in the payload
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source W

Signal	WM_INT::Pixel Shader Uses Input Coverage Mask
Description	This bit, if ENABLED, indicates that the PS kernel requires the input coverage mask to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Input Coverage Mask

Signal	WM_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table below: WM_INT::Multisample Rasterization Mode

WM_INT::Multisample Rasterization Mode

3DSTATE_RASTER:: Force Multisampling	Force	Force	Force	Force	Normal	Normal	Normal
3DSTATE_RASTER:: DX Multisample Rasterization Mode	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 Multisample Modes/State)

Note: OFF_PIXEL, OFF_PATTERN, ON_PIXEL, ON_PATTERN modes are described in Multisample Modes/State.

Signal	WM_INT::Multisample Dispatch Mode
Description	This bit, determines how PS threads are dispatched
Formula	= (WM_INT::RT Independent Rasterization Enable)? PerPixel: (3DSTATE_PS_EXTRA::PixelShaderIsPerSample) ? PerSample :PerPixel

Signal	WM_INT::Pixel Shader Kill Pixel
Description	This bit, if ENABLED, indicates that the PS kernel or color calculator has the ability to kill (discard) pixels or samples, <i>other than due to depth or stencil testing</i> .
Formula	<pre> = (3DSTATE_WM::ForceKillPixel == ON) ((3DSTATE_WM::ForceKillPixel != Off) && ! WM_INT::WM_HZ_OP && ! 3DSTATE_WM::EDSC_Mode == 2 && (WM_INT::Depth Write Enable WM_INT::Stencil Write Enable) && (3DSTATE_PS_EXTRA::PixelShaderKillsPixels 3DSTATE_PS_EXTRA:: oMask Present to RenderTarget 3DSTATE_PS_BLEND::AlphaToCoverageEnable 3DSTATE_PS_BLEND::AlphaTestEnable 3DSTATE_WM_CHROMAKEY::ChromaKeyKillEnable)) </pre>

Signal	WM_INT::Early Depth/Stencil Control
Description	This field specifies the behavior of early depth/stencil test.
Formula	<pre> = (WM_INT::WM_HZ_OP) ? EDSC_NORMAL : WM_INT::RT Independent Rasterization Enable ? EDSC_PSEXEC : 3DSTATE_WM::Early Depth/Stencil Control </pre>

Signal	WM_INT::RT Independent Rasterization Enable
Description	Enables Render Target Independent Rasterization.
Formula	<pre> = (WM_INT::WM_HZ_OP ? Disable : (3DSTATE_RASTER::ForceSampleCount != NUMRASTSAMPLES_0) ? Enable : Disable </pre>

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.

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; }

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 BDW::Depth Clear Value	Must be programmed according to the API when 3DSTATE_WM_HZ_OP::Depth Buffer Clear is set
3DSTATE_CLEAR_PARAMS BDW::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) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL
SF_INT::Cull Mode	CULLMODE_BACK	CULLMODE_BACK	CULLMODE_BACK	CULLMODE_BACK
SF_INT::Scissor Rectangle Enable	3DSTATE_WM_HZ_OP::Scissor Rectangle Enable	3DSTATE_WM_HZ_OP::Scissor Rectangle Enable	3DSTATE_WM_HZ_OP::Scissor Rectangle Enable	3DSTATE_WM_HZ_OP::Scissor Rectangle Enable
SF_INT::RT Independent Rasterization Enable	Disable	Disable	Disable	Disable
SF_INT::FrontFace Fill Mode	SOLID	SOLID	SOLID	SOLID
SF_INT::FrontWinding	FRONTWINDING_CW	FRONTWINDING_CW	FRONTWINDING_CW	FRONTWINDING_CW
SF_INT::Render Target Array index	0	0	0	0
SF_INT::Viewport index	0	0	0	0
SF_INT:: Geometry Hashing Disable	Disable	Disable	Disable	Disable
WM_INT::StencilTestEnable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::StencilWriteEnable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::DepthTest Enable	Disable	Disable	Enable	Disable
WM_INT::DepthWriteEnable	Depth buffer Clear ? Enable : Disable	Enable	Enable	Enable
WM_INT::DepthTest Function	NEVER	NEVER	NEVER	NEVER
WM_INT::StencilTestFunction	ALWAYS	Stencil buffer Clear ? ALWAYS: No Override	No Override	No Override

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable	Hierarchical Depth Buffer Resolve Enable
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::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::S ample Mask	3DSTATE_WM_HZ_OP::S ample Mask	3DSTATE_WM_HZ_OP::S ample Mask	3DSTATE_WM_HZ_OP::S ample Mask
WM_INT::Early Depth/Stencil Control	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL
WM_INT:: Full Surface Depth Clear	Depth buffer clear ? WM_HZ_OP:: Full Surface Depth Clear : Disable	WM_HZ_OP:: Full Surface Depth Clear	Disable	Disable
WM_INT:: Full Surface Depth Clear	Depth buffer clear ? WM_HZ_OP:: Full Surface Depth Clear : Disable	WM_HZ_OP:: Full Surface Depth Clear	Disable	Disable

3DSTATE_WM_DEPTH_STENCIL

Signal	WM_INT::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	$= 3DSTATE_STENCIL_BUFFER::STENCIL_BUFFER_ENABLE \&\&$ $3DSTATE_DEPTH_BUFFER::STENCIL_WRITE_ENABLE \&\&$ $($ $(WM_INT::WM_HZ_OP ?$ $\text{Use the WM_INT::StencilWriteEnable from WM_HZ_OP table :}$ $WM_INT::StencilTestEnable \&\&$ $3DSTATE_WM_DEPTH_STENCIL::StencilBufferWriteEnable$ $)$ $)$

Signal	WM_INT::StencilTestEnable
Description	Enables Stencil Test
Formula	$= 3DSTATE_STENCIL_BUFFER::STENCIL_BUFFER_ENABLE \&\&$ $($ $WM_INT::WM_HZ_OP ?$ $\text{Use the WM_INT::StencilTestEnable from WM_HZ_OP table :}$ $(3DSTATE_WM_DEPTH_STENCIL::StencilTestEnable \&\&$ $!WM_INT::RT\ Independent\ Rasterization\ Enable)$ $)$

Signal	WM_INT::DepthTestEnable
Description	Enables Depth Test
Formula	$= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE \neq NULL) \&\&$ $($ $WM_INT::WM_HZ_OP ?$ $\text{Use the WM_INT::DepthTestEnable from WM_HZ_OP table :}$ $(3DSTATE_WM_DEPTH_STENCIL::DepthTestEnable \&\&$ $! WM_INT::RT\ Independent\ Rasterization\ Enable)$ $)$

Signal	WM_INT::DepthWriteEnable
Description	Enables Depth Write
Formula	$= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE \neq NULL) \&\&$ $3DSTATE_DEPTH_BUFFER::DEPTH_WRITE_ENABLE \&\&$ $($ $WM_INT::WM_HZ_OP ?$ $\text{Use the WM_INT::DepthWriteEnable from WM_HZ_OP table :}$ $3DSTATE_WM_DEPTH_STENCIL::DepthWriteEnable$ $)$

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

Pixel 0	Pixel 1
Pixel 2	Pixel 3

B.6850-01

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* for details on the screen-space geometry of the various line types.

Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

3DSTATE_AA_LINE_PARAMS

3DSTATE_AA_LINE_PARAMETERS

The slope and bias values should be computed to closely match the reference rasterizer results Based on empirical data, the following recommendations are offered:

The final alpha for the center of the line needs to be 148 to match the reference rasterizer In this case, the Lo to edge 0 and edge 3 will be the same Since the alpha for each edge is multiplied together, we get:

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

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

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

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

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

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

Since we are using 8 bit precision, the slope becomes

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

The alpha value for Lo = 0 (second pixel from center) determines the bias term and is equal to

$$(0.211764706 - 0.550070025) = -0.338305319$$

With 8 bits of precision the programmed bias value

Line Stipple

Line stipple, controlled via the **Line Stipple Enable** state variable in WM_STATE, discards certain pixels that are produced by non-AA line rasterization.

The line stipple rule is specified via the following state variables programmed via

3DSTATE_LINE_STIPPLE: the 16-bit **Line Stipple Pattern** (p), **Line Stipple Repeat Count** I, and **Line Stipple Inverse Repeat Count**. Software must compute **Line Stipple Inverse Repeat Count** as $1.0f / \text{Line Stipple Repeat Count}$ and then converted from float to the required fixed point encoding (see 3STATE_LINE_STIPPLE).

The WM unit maintains an internal Line Stipple Counter state variable (s) The initial value of s is zero; s is incremented after production of each pixel of a line segment (pixels are produced in order, beginning at the starting point and working towards the ending point). S is reset to 0 whenever a new primitive is processed (unless the primitive type is LINESTRIP_CONT or LINESTRIP_CONT_BF), and before every line segment in a group of independent segments (LINELIST primitive).

During the rasterization of lines, the WM unit computes:

$$b = \lfloor s/r \rfloor \bmod 16,$$

A pixel is rendered if the bth bit of p is 1, otherwise it is discarded. The bits of p are numbered with 0 being the least significant and 15 being the most significant.

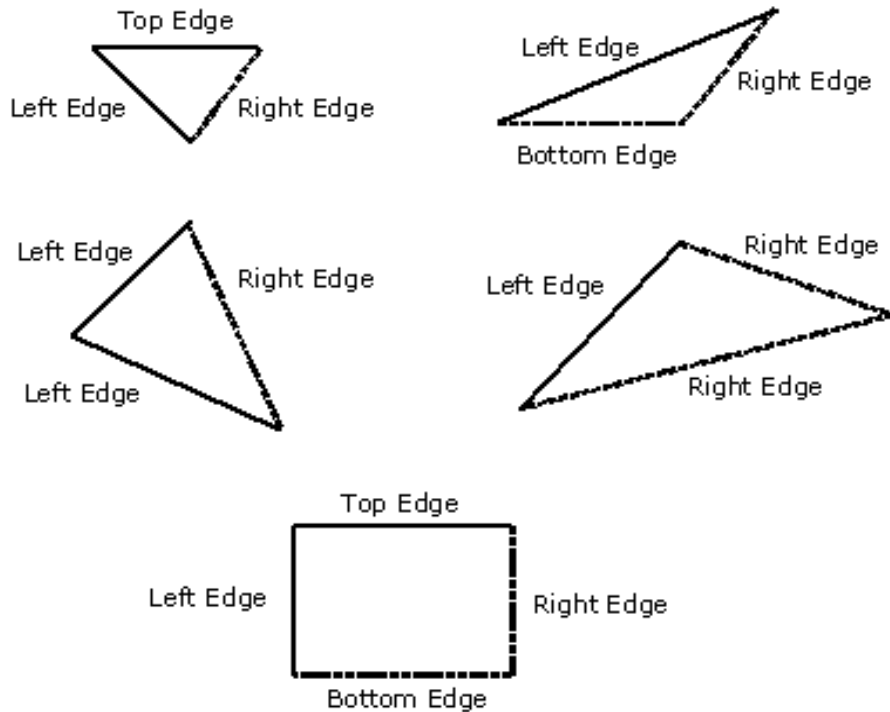
3DSTATE_LINE_STIPPLE

Polygon (Triangle and Rectangle) Rasterization

The rasterization of LINE, TRIANGLE, and RECTANGLE objects into pixels requires a "pixel sampling grid" to be defined This grid is defined as an axis-aligned array of pixel sample points spaced exactly 1 pixel unit apart If a sample point falls within one of these objects, the pixel associated with the sample point is considered "inside" the object, and information for that pixel is generated and passed down the pipeline

For TRIANGLE and RECTANGLE objects, if a sample point intersects an edge of the object, the associated pixel is considered "inside" the object if the intersecting edge is a "left" or "top" edge (or, more exactly, the intersected edge is not a "right" or "bottom" edge) Note that "top" and "bottom" edges are by definition exactly horizontal. See *TRIANGLE and RECTANGLE Edge Types* below for the edge types for representative TRIANGLE and RECTANGLE objects (solid edges are inclusive, dashed edges are exclusive).

TRIANGLE and RECTANGLE Edge Types



B.6851-01

Polygon Stipple

The *Polygon Stipple* function, controlled via the **Polygon Stipple Enable** state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered. Polygon stipple is applied only to the following primitive types:

3DPRIM_POLYGON
3DPRIM_TRIFAN
3DPRIM_TRILIST
3DPRIM_TRISTRIP
3DPRIM_TRISTRIP_REVERSE

Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command. The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

3DSTATE_POLY_STIPPLE_OFFSET

3DSTATE_POLY_STIPPLE_PATTERN

Multisampling

The multisampling function has two components:

- **Multisample Rasterization:** multisample rasterization occurs at a subpixel level, wherein each pixel consists of a number of "samples" at state-defined positions within the pixel footprint. Coverage of the primitive as well as color calculator operations (stencil test, depth test, color buffer blending, etc.) are done at the sample level. In addition the pixel shader itself can optionally run at the sample level depending on a separate state field.
- **Multisample Render Targets (MSRT):** The render targets, as well as the depth and stencil buffers, now have the ability to store per-sample values. When combined with multisample rasterization, color calculator operations such as stencil test, depth test, and color buffer blending are done with the destination surface containing potentially different values per sample.

3DSTATE_MULTISAMPLE

Signal	WM_MULTISAMPLE_INT::Number of Multisamples
Description	Set the number of multisamples
Formula	$= (\text{WM_INT}::\text{WM_HZ_OP}) ?$ 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
	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.
	WM_INT::Multisample Dispatch Mode	
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.
	3DSTATE_RASTER::ForcedSampleCount	
	3DSTATE_RASTER::ForcedSampleCount != NUMRASTSAMPLES_0	
Pixel Location	3DSTATE_MULTISAMPLE	Indicates the subpixel location where values specified as "pixel" are sampled. This is either the upper left corner or the center.
	3DSTATE_SAMPLE_PATTERN	
	3DSTATE_SAMPLE_PATTERN	
API Mode	3DSTATE_RASTER	One of the deciding factors of what the Multisample Rasterization Mode should be according to WM_INT::Multisample Rasterization Mode. Software sets this field according to the API's version.
DX Multisample Rasterization Enable	3DSTATE_RASTER	Controls ON/OFF part of Multisample Rasterization Mode, depending on the API Mode according to WM_INT::Multisample Rasterization Mode.

This table does not apply if (3DSTATE_RASTER::ForceMultisampleRasterMode == Force) or (WM_INT::WM_HZ_OP == true).

Table 1: Multisample Rasterization Modes

Number of Multisamples	NUMSAMP LES_1	NUMSAMPL ES_1	> NUMSAMPL ES_1	> NUMSAMPL ES_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	NUMRA ST SAMPLE S_1	NUMRA ST SAMPLE S_1	> NUMRAST SAMPLES_ 1	> NUMRAST SAMPLES_ 1
API Mode == DX9.0/OpenGL	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_PAT ERN	ON_PAT ERN

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.
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-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

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
		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

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:
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:

$\text{MaxDepthSlope} = \max(\text{abs}(dZ/dX), \text{abs}(dz/dy))$ // approximation of max depth slope for polygon

When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

$$\text{Bias} = \text{GlobalDepthOffsetConstant} * r + \text{GlobalDepthOffsetScale} * \text{MaxDepthSlope}$$

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:

$$\text{Bias} = \text{GlobalDepthOffsetConstant} * 2^{(\text{exponent}(\text{max } z \text{ in primitive}) - r) + \text{GlobalDepthOffsetScale} * \text{MaxDepthSlope}$$

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 GlobalDepthOffsetConstant).

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
```

Biassing 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 value of Z_Height and Z_Width must each be multiplied by 2 before being applied to the table below if **Number of Multisamples** is set to NUMSAMPLES_4. The value of Z_Height must be multiplied by 2 and Z_Width must be multiplied by 4 before being applied to the table below if **Number of Multisamples** is set to NUMSAMPLES_8.

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	ceiling(Z_Width / 16) * 16	see below	not applicable
SURFTYPE_CUBE	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/2)/8) * 8 * 6	see below

To compute the minimum QPitch for the HZ surface, the height of each LOD in pixels is determined using the equations for h_L 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:

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

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:

$$HZ_Height = \frac{1}{2} \left[\sum_{i=0}^m h_i * \text{floor} \left(\frac{Z_Depth}{2^i} \right) \right]$$

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 format is D32_FLOAT, then +/-DENORM values are also allowed.
 - The following alignment restrictions need to be met while doing the fast-clear:

The following restrictions apply only if the depth buffer surface type is D16_UNORM and software does not use the “full surf clear”:

If **Number of Multisamples** is NUMSAMPLES_1, the rectangle must be aligned to an 8x4 pixel block relative to the upper left corner of the depth buffer, and contain an integer number of these pixel blocks, and all 8x4 pixels must be lit.

If **Number of Multisamples** is NUMSAMPLES_2, the rectangle must be aligned to a 4x4 pixel block (8x4 sample block) relative to the upper left corner of the depth buffer, and contain an integer number of these pixel blocks, and all samples of the 4x4 pixels must be lit.

If **Number of Multisamples** is NUMSAMPLES_4, the rectangle must be aligned to a 4x2 pixel block (8x4 sample block) relative to the upper left corner of the depth buffer, and contain an integer number of these pixel blocks, and all samples of the 4x2 pixels must be lit.

If **Number of Multisamples** is NUMSAMPLES_8, the rectangle must be aligned to a 2x2 pixel block (8x4 sample block) relative to the upper left corner of the depth buffer, and contain an integer number of these pixel blocks, and all samples of the 2x2 pixels must be lit.

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 8x4 pixel aligned (if MSAA enabled, 4x4 for 2X, 4x2 for 4X, 2x2 for 8X), 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 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 next rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

The following is required when performing a depth buffer resolve:

- The surface must have been initialized with a Depth Buffer Clear after its allocation to initialize the Depth Clear Value.
- A rectangle primitive of the same size as the previous depth buffer clear operation must be delivered, and depth buffer state cannot have changed since the previous depth buffer clear operation.
- **Depth Test Enable** must be enabled with the **Depth Test Function** set to NEVER. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch**, **Alpha Test**, **Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

Programming Note

HW uses the clear value from the 3DSTATE_CLEAR_PARAM. If you change the value in the 3DSTATE_CLEAR_PARAMS before resolve, it will flush the depth caches and have the new-clear value in its register. When doing the resolve pass, it is driver’s responsibility to make sure that the clear-value for the depth buffer is the same one as the clear-pass.

Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation. If this occurs, the hierarchical depth buffer must be *resolved* to avoid incorrect device behavior. This is done by setting the Hierarchical Depth Buffer Resolve Enable field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be delivered.
- **Depth Test Enable** must be disabled. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch**, **Alpha Test**, **Pixel Shader Kill Pixel**, and **Pixel Shader Computed Depth** must all be disabled.

Optimized Depth Buffer Clear and/or Stencil Buffer Clear

With the hierarchical depth buffer enabled, performance is generally improved by using the special clear mechanism described here to clear the hierarchical depth buffer and the depth buffer. This is enabled through the **Depth Buffer Clear** field in 3DSTATE_WM_HZ_OP. This bit can be used to clear the depth buffer in the following situations:

- All 3D units before SF will be bypassed by WM_HZ_OP and states for those units need not be set/restored for these rectangles.
- Complete depth buffer clear
- Partial depth buffer clear with the clear value the same as the one used on the previous clear
- Partial depth buffer clear with the clear value different than the one used on the previous clear can use this mechanism if a depth buffer resolve is performed first.
- The minimum granularity of clear is one pixel, but all samples of the pixel must be cleared. Clearing partial samples of a pixel is not supported.

Stencil Buffer Clears can be alone or at the same time as depth buffer clears by using the Stencil Buffer Clear bit in 3DSTATE_WM_HZ_OP.

The proper sequence of commands is as follows:

1. Setup 3DSTATE_DEPTH_BUFFER (as needed). Render Target Array index will be internally force to zero. SW must set 3DSTATE_DEPTH_BUFFER::**MinimumArrayElement** to render to the array to be cleared.
2. Setup 3DSTATE_HIER_DEPTH_BUFFER (as needed)
3. Setup 3DSTATE_STENCIL_BUFFER (as needed)
4. Setup 3DSTATE_CLEAR_PARMS (as needed)
5. Setup 3DSTATE_DRAWING_RECTANGLE (as needed and only if it is different from already existing drawing rectangle)
6. 3DSTATE_WM_HZ_OP w/ 1 of the clear/resolve bits set
// This overrides existing state and forces them to what is needed for the clear
// This also carries the vertex info for doing the clear
7. PIPE_CONTROL w/ all bits clear except for "Post-Sync Operation" must set to "Write Immediate Data" enabled.
// This causes 3DSTATE_WM_HZ_OP state to be committed to SF and WM as a pipeline state. Once state is committed to SF, causes to spawn a rectangle to be drawn
8. 3DSTATE_WM_HZ_OP w/ none of the clear/resolve bits set
// This clears the overrides
9. Restore 3DSTATE_DEPTH_BUFFER (as needed).
10. Restore 3DSTATE_HIER_DEPTH_BUFFER (as needed)
11. Restore 3DSTATE_STENCIL_BUFFER (as needed)

Arbitrary size rectangles are supported using the Top Left X, Top Left Y, Bottom Right X, Bottom Right Y fields in the 3DSTATE_WM_HZ_OP.

Optimized Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete. If the depth buffer is retained and used for another purpose (i.e. locked by the app), it must first be “resolved”. This is done by setting the **Depth Buffer Resolve Enable** field in

3DSTATE_WM_HZ_OP. The depth buffer resolve uses the same sequence as the optimized Depth buffer clear (see above) except the **Depth Buffer Resolve Enable** bit is set. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a source in three different cases: using it as a texture for the next rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

Doing a resolve operation requires that a preceding Depth Buffer Clear operation is required to have initialized the Depth Clear Value.

Optimized Hierarchical Depth Buffer Resolve

For BDW, 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.

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

3DSTATE_PS_EXTRA

This command is used to set state used by the pixel shader dispatch stage.

3DSTATE_PS_BLEND

This command is used to set state used by the pixel shader dispatch stage.

3DSTATE_CONSTANT_PS

3DSTATE_BINDING_TABLE_POINTERS_PS

3DSTATE_PUSH_CONSTANT_ALLOC_PS

3DSTATE_SAMPLER_STATE_POINTERS_PS

Pixel Grouping (Dispatch Size) Control

The WM unit can pass a grouping of 2 subspans (8 pixels), 4 subspans (16 pixels), or 8 subspans (32 pixels) to a Pixel Shader thread. Software should take into account the following considerations when determining which groupings to support/enable during operation. This determination involves a tradeoff of these likely conflicting issues. Note that the size of the dispatch has significant impact on the kernel program. (It is certainly not transparent to the kernel.) Also note that there is no implied spatial relationship between the subspans passed to a PS thread, other than the fact that they come from the same object.

- **Thread Efficiency:** In general, there is some amount of overhead involved with PS thread dispatch, and if this can be amortized over a larger number of pixels, efficiency will likely increase. This is especially true for very short PS kernels, as may be used for desktop composition, etc.
- **GRF Consumption:** Processing more pixels per thread requires a larger thread payload and likely more temporary register usage, both of which translate into a requirement for a larger GRF register allocation for the threads. This increased GRF usage could lead to increased use of scratch space (for spill/fill, etc.) and possibly less efficient use of the EUs (as it would be less likely to find an EU with enough free physical GRF registers to service the thread).
- **Object Size:** If the number of very small objects (e.g., covering 2 subspans or fewer) is expected to comprise a significant portion of the workload, supporting the 8-pixel dispatch mode may be advantageous. Otherwise there could be a large number of 16-pixel dispatches with only 1 or 2 valid subspans, resulting in low efficiency for those threads.
- **Intangibles:** Kernel footprint & Instruction Cache impact; Complexity;

The groupings of subspans that the WM unit is allowed to include in a PS thread payload is controlled by the **32,16,8 Pixel Dispatch Enable** state variables programmed in WM_STATE. Using these state variables, the WM unit attempts to dispatch the largest allowed grouping of subspans. The following table lists the possible combinations of these state variables.

Please note that, the valid column in the table indicates which products supports the combination dispatch. Combinations that are not listed in the table are not available on any product.

The letter codes A, B, D, and E used in the Variable Pixel Dispatch table below are valid for all projects with some specific mode restrictions for specific projects for B, D, and E as indicated in the next few tables.

D is like B with an added general restriction, that it cannot be used in non-1x PERSAMPLE mode.

E cannot be used in PERSAMPLE mode with number of multisamples ≥ 2 .

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.

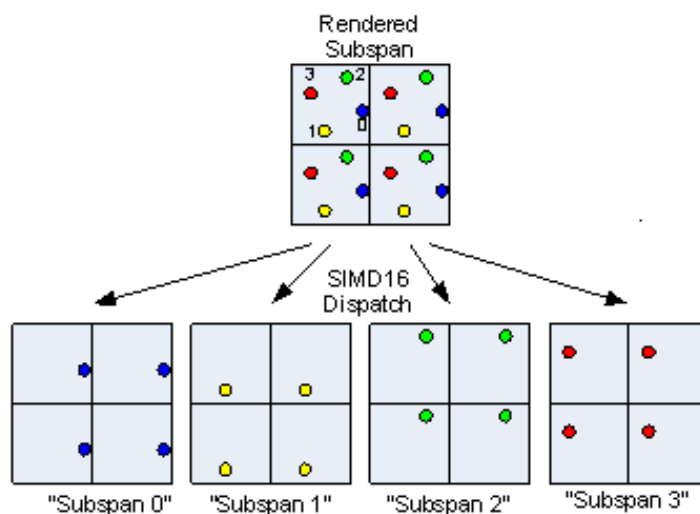
MSDISPMODE_PERPIXEL Thread Dispatch

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.

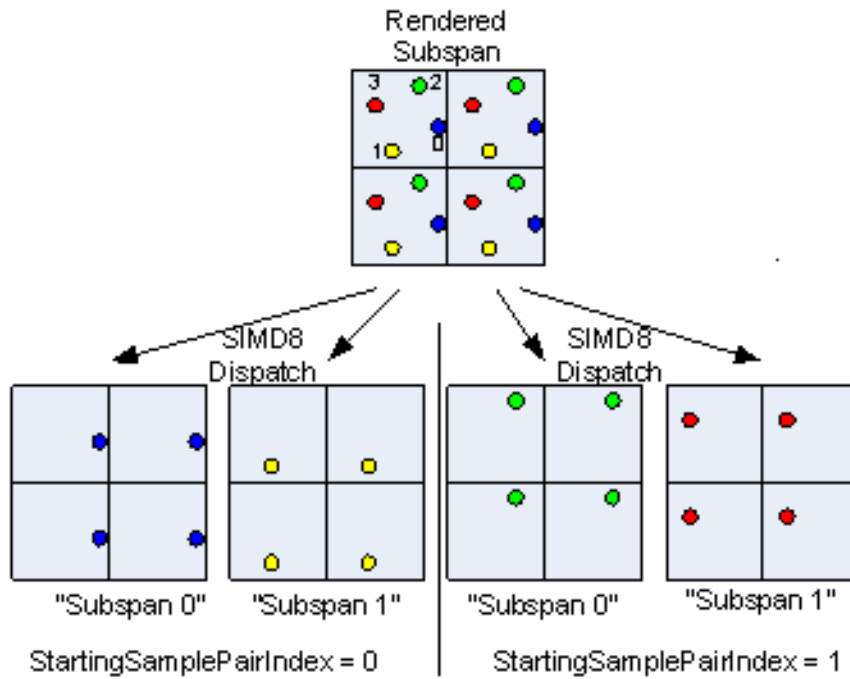
MSDISPMODE_PERSAMPLE Thread Dispatch

In PERSAMPLE mode, the pixel shader needs to operate on a sample vs. pixel basis (although this collapses in NUMSAMPLES_1 mode) Instead of processing strictly different subspans in parallel, the PS kernel processes different sample indices of one or more subspans in parallel For example, a SIMD16 dispatch in PERSAMPLE/NUMSAMPLES_4 mode would operate on a single subspan, with the usual "4 Subspan0 pixel slots" used for the "4 Sample0 locations of the (single) subspan" Subspan1 slots would be used for the Sample1 locations, and so on This layout allows the pixel shader to compute derivatives/LOD based on deltas between corresponding sample locations in the subspan in the same fashion as LEGACY pixel shader execution, and as required by DX10.1.

Depending on the dispatch mode (8/16/32 pixels) and multisampling mode (1X/4X), there are different mappings of subspans/samples onto dispatches and slots-within-dispatch In some cases, more than one subspan may be included in a dispatch, while in other cases multiple dispatches are required to process all samples for a single subspan In the latter case, the **StartingSamplePairIndex** value is included in the payload header so the Render Target Write message will access the correct samples with each message.



PERSAMPLE SIMD16 4X Dispatch



PERSAMPLE Dispatch

The following table provides the complete dispatch/slot mappings for all the MS/Dispatch combinations.

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
SIMD32	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0] Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0] Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0] Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]
	2X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1] Slot[19:16] = Subspan[2].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[2].Pixel[3:0].Sample[1] Slot[27:24] = Subspan[3].Pixel[3:0].Sample[0] Slot[31:28] = Subspan[3].Pixel[3:0].Sample[1]
	4X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3] Slot[19:16] = Subspan[1].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[1].Pixel[3:0].Sample[1] Slot[27:24] = Subspan[1].Pixel[3:0].Sample[2] Slot[31:28] = Subspan[1].Pixel[3:0].Sample[3]
	8X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] 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]
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]

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

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

The following payload table applies to **BDW**. 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 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
R0.2	31:0	Reserved: Delivered as zeros (reserved for message header fields).
R0.1	31:6	Color Calculator State Pointer: Specifies the 64-byte aligned pointer to the Color Calculator state (COLOR_CALC_STATE structure in memory). It is specified as an offset from the Dynamic State Base Address . This value is eventually passed to the ColorCalc function in the DataPort and is used to fetch the corresponding CC_STATE data. Format = DynamicStateOffset[31:5]
	5:0	Reserved
R0.0	31	Reserved

DWord	Bits	Description														
	30:27	Viewport Index: Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]														
	26:16	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	Face	Render Target Array Index	+x	0	-x	1	+y	2	-y	3	+z	4	-z	5
Face	Render Target Array Index															
+x	0															
-x	1															
+y	2															
-y	3															
+z	4															
-z	5															
	15	Front/Back Facing Polygon: Determines whether the polygon is front or back facing. Used by 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 <i>3D Pipeline</i> .)														
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.														

DWord	Bits	Description
	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
R1.4	31:16	Y2: Y coordinate (screen space) for upper-left pixel of subspan 2 (slot 8). Format = U16
	15:0	X2: X coordinate (screen space) for upper-left pixel of subspan 2 (slot 8). Format = U16
R1.3	31:16	Y1: Y coordinate (screen space) for upper-left pixel of subspan 1 (slot 4). Format = U16
	15:0	X1: X coordinate (screen space) for upper-left pixel of subspan 1 (slot 4). Format = U16
R1.2	31:16	Y0: Y coordinate (screen space) for upper-left pixel of subspan 0 (slot 0). Format = U16
	15:0	X0: X coordinate (screen space) for upper-left pixel of subspan 0 (slot 0). Format = U16
R1.1	31:0	Reserved
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 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]

DWord	Bits	Description
	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
	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]

DWord	Bits	Description
	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] 16X MSAA range [0..15]
		R3-R26: Delivered only if the corresponding Barycentric Interpolation Mode bit is set. Register phases containing Slot 8-15 data are not delivered in <i>8-pixel dispatch</i> mode.
R3.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 7 This and the next register phase is only included if the corresponding enable bit in Barycentric Interpolation Mode is set. Format = IEEE_Float
R3.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 6
R3.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 5
R3.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 4
R3.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 3
R3.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 2
R3.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 1
R3.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 0
R4		Perspective Pixel Location Barycentric[2] for Slots 7:0
R5.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 15
R5.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 14
R5.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 13
R5.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 12
R5.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 11
R5.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 10

DWord	Bits	Description
R5.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 9
R5.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 8
R6		Perspective Pixel Location Barycentric[2] for Slots 15:8
R7:10		Perspective Centroid Barycentric
R11:14		Perspective Sample Barycentric
R15:18		Linear Pixel Location Barycentric
R19:22		Linear Centroid Barycentric
R23:26		Linear Sample Barycentric
		R27: Delivered only if Pixel Shader Uses Source Depth is set.
R27.7	31:0	Interpolated Depth for Slot 7 Format = IEEE_Float This and the next register phase is only included if Pixel Shader Uses Source Depth (WM_STATE) is set.
R27.6	31:0	Interpolated Depth for Slot 6
R27.5	31:0	Interpolated Depth for Slot 5
R27.4	31:0	Interpolated Depth for Slot 4
R27.3	31:0	Interpolated Depth for Slot 3
R27.2	31:0	Interpolated Depth for Slot 2
R27.1	31:0	Interpolated Depth for Slot 1
R27.0	31:0	Interpolated Depth for Slot 0
		R28: Delivered only if Pixel Shader Uses Source Depth is set and this is not an <i>8-pixel dispatch</i> .
R28.7	31:0	Interpolated Depth for Slot 15
R28.6	31:0	Interpolated Depth for Slot 14
R28.5	31:0	Interpolated Depth for Slot 13
R28.4	31:0	Interpolated Depth for Slot 12
R28.3	31:0	Interpolated Depth for Slot 11
R28.2	31:0	Interpolated Depth for Slot 10
R28.1	31:0	Interpolated Depth for Slot 9
R28.0	31:0	Interpolated Depth for Slot 8
		R29: Delivered only if Pixel Shader Uses Source W is set.
R29.7	31:0	Interpolated W for Slot 7 Format = IEEE_Float 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

DWord	Bits	Description
R29.2	31:0	Interpolated W for Slot 2
R29.1	31:0	Interpolated W for Slot 1
R29.0	31:0	Interpolated W for Slot 0
		R30: Delivered only if Pixel Shader Uses Source W is set and this is not an <i>8-pixel dispatch</i> .
R30.7	31:0	Interpolated W for Slot 15
R30.6	31:0	Interpolated W for Slot 14
R30.5	31:0	Interpolated W for Slot 13
R30.4	31:0	Interpolated W for Slot 12
R30.3	31:0	Interpolated W for Slot 11
R30.2	31:0	Interpolated W for Slot 10
R30.1	31:0	Interpolated W for Slot 9
R30.0	31:0	Interpolated W for Slot 8
		R31: Delivered only if Position XY Offset Select is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE.
R31.7	31:24	Position Offset Y for Slot 15 This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of Position XY Offset Select . Format = U4.4 Range = [0.0,1.0)
	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 Range = [0.0,1.0)
	15:8	Position Offset Y for Slot 14
	7:0	Position Offset X for Slot 14
R31.6	31:24	Position Offset Y for Slot 13
	23:16	Position Offset X for Slot 13
	15:8	Position Offset Y for Slot 12
	7:0	Position Offset X for Slot 12
R31.5:4		Position Offset X/Y for Slot[11:8]
R31.3:2		Position Offset X/Y for Slot[7:4]
R31.1:0		Position Offset X/Y for Slot[3:0]
		R32: Delivered only if Pixel Shader Uses Input Coverage Mask is set.
R32.7	31:0	Input Coverage Mask for Slot 7 Format = U32 This and the next register phase is only included if Pixel Shader Uses Input Coverage Mask (3DSTATE_PS) is set. This field always encodes sample Coverage Mask.

DWord	Bits	Description
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 <i>8-pixel dispatch</i> .
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 <i>32-pixel dispatch</i> .
R34.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 23 This and the next register phase is only included if the corresponding enable bit in Barycentric Interpolation Mode is set. Format = IEEE_Float
R34.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 22
R34.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 21
R34.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 20
R34.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 19
R34.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 18
R34.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 17
R34.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 16
R35		Perspective Pixel Location Barycentric[2] for Slots 23:16
R36.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 31
R36.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 30
R36.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 29
R36.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 28
R36.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 27
R36.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 26
R36.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 25

DWord	Bits	Description
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.
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

DWord	Bits	Description
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
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

DWord	Bits	Description
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

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 BDW 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.

OpenGL defines a lower-left screen coordinate origin. This requires the driver to incorporate a "Y coordinate flipping" transformation into the viewport mapping function. The Y extent of the RT is used in this flipping transformation. If the DepthBuffer extent is different, the wrong pixel Y locations within the DepthBuffer will be accessed.

The least expensive solution is to provide a translation offset to be applied to the post-viewport-mapped DepthBuffer Y pixel coordinate, effectively allowing the alignment of the lower-left origins of the RT and DepthBuffer. [Note that the previous DBCOD feature performed an optional translation of post-viewport-mapping RT pixel (screen) coordinates to generate DepthBuffer pixel (window) coordinates. Specifically, the Draw Rect Origin X,Y state could be subtracted from the RT pixel coordinates.]

This function uses **Depth Coordinate Offset X,Y** state (signed 16-bit values in 3DSTATE_DEPTH_RECTANGLE) that is unconditionally added to the RT pixel coordinates to generate DepthBuffer pixel coordinates.

The previous DBCOB feature can be supported by having the driver program Depth Coordinate X,Y Offset to the two's complement of the the Draw Rect Origin. By programming Depth Coordinate X,Y Offset to zeros, the current "normal" operation (DBCOD disabled) can be achieved.

Programming Note	
Context:	Depth Coordinate Offset
<ul style="list-style-type: none"> Only simple 2D RTs are supported (no mipmaps). Software must ensure that the resultant DepthBuffer Coordinate X,Y values are non-negative. There are alignment restrictions – see 3DSTATE_DEPTH_BUFFER command.on SFID_DP_DC2) are IA coherent. 	

Stencil Test

The Stencil Test function can be used to discard pixels based on a comparison between the **[Backface] Stencil Test Reference** state variable and the pixel's stencil value. This is a general purpose function used for such effects as shadow volumes, per-pixel clipping, etc. The result of this comparison is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Stencil Test Enable** state variable. If ENABLED, the current stencil buffer value for this pixel is read.

A 2nd set of the stencil test state variables is provided so that pixels from back-facing objects, assuming they are not culled, can have a stencil test performed on them separate from the test for normal front-facing objects. The separate stencil test for back-facing objects can be enabled via the **Double Sided Stencil Enable** state variable. Otherwise, non-culled back-facing objects will use the same test function, mask and reference value as front-facing objects. The 2nd stencil state for back-facing objects is most commonly used to improve the performance of rendering shadow volumes which require a different stencil buffer operation depending on whether pixels rendered are from a front-facing or back-facing object. The backface stencil state removes the requirement to render the shadow volumes in 2 passes or sort the objects into front-facing and back-facing lists.

The remainder of this subsection describes the function in term of **[Backface] <state variable name>**. The Backface set of state variables are only used if Double Sided Stencil Enable is ENABLED and the object is considered back-facing. Otherwise the normal (front-facing) state variables are used.

This function then compares the **[Backface] Stencil Test Reference** value and the pixel's stencil value value after logically ANDing both values by **[Backface] Stencil Test Mask**. The comparison performed is specified by the **[Backface] Stencil Test Function** state variable. The result of the comparison is passed down the pipeline for use in the Stencil Buffer Update function. The Stencil Test function does not in itself discard pixels.

If **Stencil Test Enable** is DISABLED, a result of "stencil test passed" is propagated down the pipeline.

Depth Test

The Depth Test function can be used to discard pixels based on a comparison between the incoming pixel's depth value and the current depth buffer value associated with the pixel. This function is typically used to perform the "Z Buffer" hidden surface removal. The result of this pipeline function is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Depth Test Enable** state variable. If enabled, the pixel's ("source") depth value is first computed. After computation the pixel's depth value is clamped to the range defined by **Minimum Depth** and **Maximum Depth** in the selected CC_VIEWPORT state. Then the current ("destination") depth buffer value for this pixel is read.

This function then compares the source and destination depth values. The comparison performed is specified by the **Depth Test Function** state variable.

The result of the comparison is propagated down the pipeline for use in the subsequent Depth Buffer Update function. The Depth Test function does not in itself discard pixels.

If **Depth Test Enable** is DISABLED, a result of "depth test passed" is propagated down the pipeline.

Programming Note:

- Enabling the Depth Test function without defining a Depth Buffer is UNDEFINED.

Pre-Blend Color Clamping

Pre-Blend Color Clamping, controlled via **Pre-Blend Color Clamp Enable** OR **Pre-Blend Source Only Clamp Enable** and **Color Clamp Range** states in COLOR_CALC_STATE, is affected by the enabling of Color Buffer Blend as described below.

The following table summarizes the requirements involved with Pre-/Post-Blend Color Clamping.

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
<ul style="list-style-type: none"> Given the possibility of writing UNPREDICTABLE values to the Color Buffer, it is expected and highly recommended that, when blending is disabled, software set Pre-Blend Color Clamp Enable to ENABLED and select an appropriate Color Clamp Range. When using SINT or UINT rendertarget surface formats, Blending must be DISABLED. The Pre-Blend Color Clamp Enable and Color Clamp Range fields are ignored, and an implied clamp to the rendertarget surface format is performed. 	

Pre-Blend Color Clamping When Blending is Enabled

The clamping of source, destination and constant color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all these color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed on these color components prior to blending.

Color Buffer Blending

The Color Buffer Blending function is used to combine one or two incoming “source” pixel color+alpha values with the “destination” color+alpha read from the corresponding location in a RenderTarget.

Blending is enabled on a global basis by the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE). If DISABLED, Blending and Post-Blend Clamp functions are disabled for all RenderTargets, and the pixel values (possibly subject to Pre-Blend Clamp) are passed through unchanged.

The Color Buffer Blend Enable is in the per-render-target BLEND_STATE, and the field in SURFACE_STATE is no longer supported.

Programming Note	
Context:	Color Buffer Blending, Logic Ops, DataPort, surface formats, render targets
<ul style="list-style-type: none"> Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED. Dual source blending: The DataPort only supports dual source blending with a SIMD8-style message. Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in <i>Sampling Engine</i>. Blending must be disabled on a RenderTarget if blending is not supported. 	

The incoming “source” pixel values are modulated by a selected “source” blend factor, and the possibly gamma-decorrected “destination” values are modulated by a “destination” blend factor. These terms are then combined with a “blend function”. In general:

$$\text{src_term} = \text{src_blend_factor} * \text{src_color}$$

$\text{dst_term} = \text{dst_blend_factor} * \text{dst_color}$

$\text{color output} = \text{blend_function}(\text{src_term}, \text{dst_term})$

If there is no alpha value contained in the Color Buffer, a default value of 1.0 is used and, correspondingly, there is no alpha component computed by this function.

Dual Source Blending: When using “Dual Source” Render Target Write messages, the Source1 pixel color+alpha passed in the message can be selected as a src/dst blend factor. See [Color Buffer Blend Color Factors](#). In single-source mode, those blend factor selections are invalid. If SRC1 is included in a src/dst blend factor and a DualSource RT Write message is not used, results are UNDEFINED. (This reflects the same restriction in DX APIs, where undefined results are produced if “o1” is not written by a PS – there are no default values defined). If SRC1 is not included in a src/dst blend factor, dual source blending must be disabled.

The blending of the color and alpha components is controlled with two separate (color and alpha) sets of state variables. However, if the **Independent Alpha Blend Enable** state variable in COLOR_CALC_STATE is DISABLED, then the “color” (rather than “alpha”) set of state variables is used for both color and alpha. Note that this is the only use of the **Independent Alpha Blend Enable** state – it does not control whether Blending occurs, only how.

Per **Render Target Blend State:** Blend state is selected based on **Render Target Index** contained in the message header, and appropriate blend state is applied to Render Target Write messages.

The following table describes the color source and destination blend factors controlled by the **Source [Alpha] Blend Factor** and **Destination [Alpha] Blend Factor** state variables in COLOR_CALC_STATE. Note that the blend factors applied to the R,G,B channels are always controlled by the **Source/Destination Blend Factor**, while the blend factor applied to the alpha channel is controlled either by **Source/Destination Blend Factor** or **Source/Destination Alpha Blend Factor**.

Color Buffer Blend Color Factors

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2 nd output from PS in Dual-Source mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_SRC1_COLOR	o1.r, o1.g, o1.b, o1.a
BLENDFACTOR_INV_SRC1_COLOR	1.0-o1.r, 1.0-o1.g, 1.0-o1.b, 1.0-o1.a
BLENDFACTOR_SRC1_ALPHA	o1.a, o1.a, o1.a, o1.a
BLENDFACTOR_INV_SRC1_ALPHA	1.0-o1.a, 1.0-o1.a, 1.0-o1.a, 1.0-o1.a
BLENDFACTOR_DST_COLOR	rtN.r, rtN.g, rtN.b, rtN.a
BLENDFACTOR_INV_DST_COLOR	1.0-rtN.r, 1.0-rtN.g, 1.0-rtN.b, 1.0-rtN.a
BLENDFACTOR_DST_ALPHA	rtN.a, rtN.a, rtN.a, rtN.a
BLENDFACTOR_INV_DST_ALPHA	1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a
BLENDFACTOR_CONST_COLOR	CC.r, CC.g, CC.b, CC.a
BLENDFACTOR_INV_CONST_COLOR	1.0-CC.r, 1.0-CC.g, 1.0-CC.b, 1.0-CC.a
BLENDFACTOR_CONST_ALPHA	CC.a, CC.a, CC.a, CC.a
BLENDFACTOR_INV_CONST_ALPHA	1.0-CC.a, 1.0-CC.a, 1.0-CC.a, 1.0-CC.a
BLENDFACTOR_SRC_ALPHA_SATURATE	f,f,f,1.0 where f = min(1.0 – rtN.a, oN.a)

The following table lists the supported blending operations defined by the **Color Blend Function** state variable and the **Alpha Blend Function** state variable (when in independent alpha blend mode).

Color Buffer Blend Functions

Blend Function	Operation (for each color component)
BLENDFUNCTION_ADD	SrcColor*SrcFactor + DstColor*DstFactor
BLENDFUNCTION_SUBTRACT	SrcColor*SrcFactor - DstColor*DstFactor
BLENDFUNCTION_REVERSE_SUBTRACT	DstColor*DstFactor - SrcColor*SrcFactor
BLENDFUNCTION_MIN	min (SrcColor*SrcFactor, DstColor*DstFactor) Programming Note: This is a superset of the OpenGL “min” function.
BLENDFUNCTION_MAX	max (SrcColor*SrcFactor, DstColor*DstFactor) Programming Note: This is a superset of the OpenGL “max” function.

Post-Blend Color Clamping

(See *Pre-Blend Color Clamping* above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

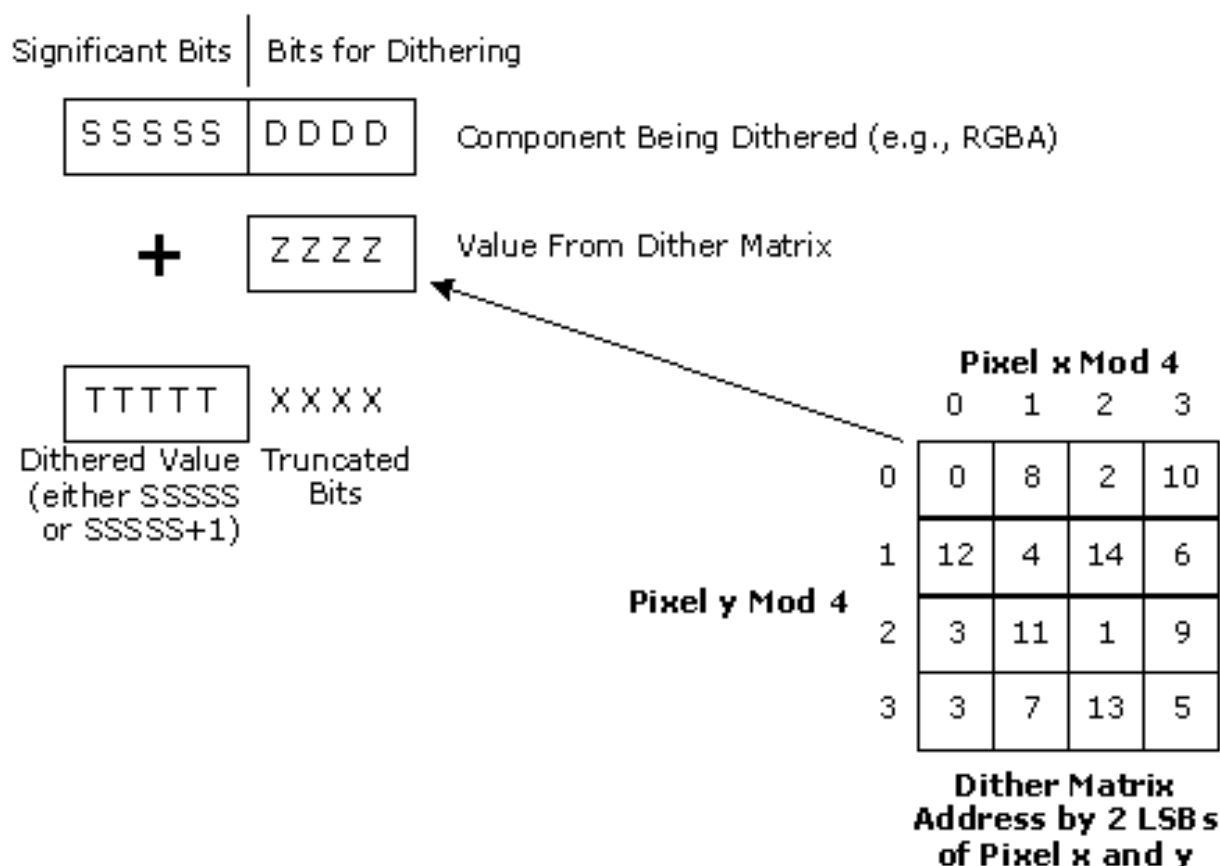
Regardless of the setting of **Post-Blend Color Clamp Enable**, when Blending is enabled color components will be automatically clamped to (at least) the rendertarget surface format range at this stage of the pipeline.

Dithering

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer's eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel's X and Y screen coordinate. The pixel's X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel's screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel's screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.



B 6852-01

Dithering Process (5-Bit Example)

Logic Ops

The Logic Ops function is used to combine the incoming "source" pixel color/alpha values with the corresponding "destination" color/alpha contained in the ColorBuffer, using a logic function.

The Logic Op function is enabled by the **LogicOp Enable** state variable. If DISABLED, this function is ignored and the incoming pixel values are passed through unchanged.

Programming Notes

Programming Note
Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
Logic Ops are supported on all blendable render targets and render targets with *INT formats.

The following table lists the supported logic ops. The logic op is selected using the **Logic Op Function** field in COLOR_CALC_STATE.

Logic Ops

LogicOp Function	Definition (S=Source, D=Destination)
LOGICOP_CLEAR	all 0's
LOGICOP_NOR	NOT (S OR D)
LOGICOP_AND_INVERTED	(NOT S) AND D
LOGICOP_COPY_INVERTED	NOT S
LOGICOP_AND_REVERSE	S AND NOT D
LOGICOP_INVERT	NOT D
LOGICOP_XOR	S XOR D
LOGICOP_NAND	NOT (S AND D)
LOGICOP_AND	S AND D
LOGICOP_EQUIV	NOT (S XOR D)
LOGICOP_NOOP	D
LOGICOP_OR_INVERTED	(NOT S) OR D
LOGICOP_COPY	S
LOGICOP_OR_REVERSE	S OR NOT D
LOGICOP_OR	S OR D
LOGICOP_SET	all 1's

Buffer Update

The Buffer Update function is responsible for updating the pixel's Stencil, Depth and Color Buffer contents based upon the results of the Stencil and Depth Test functions. Note that Kill Pixel and/or Alpha Test functions may have already discarded the pixel by this point.

Stencil Buffer Updates

If and only if stencil testing is enabled, the Stencil Buffer is updated according to the **Stencil Fail Op**, **Stencil Pass Depth Fail Op**, and **Stencil Pass Depth Pass Op** state (or their backface counterparts if **Double Sided Stencil Enable** is ENABLED and the pixel is from a back-facing object) and the results of the Stencil Test and Depth Test functions.

Stencil Fail Op and **Backface Stencil Fail Op** specify how/if the stencil buffer is modified if the stencil test fails. **Stencil Pass Depth Fail Op** and **Backface Stencil Pass Depth Fail Op** specify how/if the stencil buffer is modified if the stencil test passes but the depth test fails. **Stencil Pass Depth Pass Op** and **Backface Stencil Pass Depth Pass Op** specify how/if the stencil buffer is modified if both the stencil and depth tests pass. The operations (on the stencil buffer) that are to be performed under one of these (mutually exclusive) conditions is summarized in the following table.

Stencil Buffer Operations

Stencil Operation	Description
STENCILOP_KEEP	Do not modify the stencil buffer
STENCILOP_ZERO	Store a 0
STENCILOP_REPLACE	Store the <i>StencilTestReference</i> reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible 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

This topic is currently under development.

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)

MCS buffer can be enabled for two purposes described below. MCS buffer can be controlled using two mechanisms:

1. MMIO bit Cache Mode 1 (0x2124) register bit 5
2. RT surface state

The following table summarizes modes of operation related to MCS buffer:

Cache Mode MMIO Bit (Please refer to Vol 1c)	MSC Enable (Surface State)	Operation
1 (feature disable)	X	Normal mode of operation i.e. no MSAA compression and no color clear
0	0	Normal mode of operation i.e. no MSAA compression and no color clear
0	1	Depending on the Number of multi-samples, either MSAA compression OR color clear is enabled

MSAA	Width of Clear Rect	Height of Clear Rect
2x	Ceil($1/8 \times \text{width}$)	Ceil($1/2 \times \text{height}$)
4X	Ceil($1/8 \times \text{width}$)	Ceil($1/2 \times \text{height}$)
8X	Ceil($1/2 \times \text{width}$)	Ceil($1/2 \times \text{height}$)

- **MSAA Compression:** Multi-sample render target is bound to the pipeline and MSAA compression feature is enabled. In this case, MCS buffer stores the information required for MSAA compression algorithm. The size and layout of the MCS buffer is based on per-pixel RT. For 4X and 8X MSAA, MCS buffer element is 8bpp and 32bpp respectively. Height, width, and layout of MCS buffer in this case must match with Render Target height, width, and layout. MCS buffer is tiledY. When MCS buffer is enabled and bound to MSRT, it is required that it is cleared prior to

any rendering. A clear value can be specified optionally in the surface state of the corresponding RT. Clear pass for this case requires that scaled down primitive is sent down with upper left coordinate to coincide with actual rectangle being cleared. For MSAA, clear rectangle's height and width need to as show in the following table in terms of (width,height) of the RT.

- **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

Restrictions		
Support is limited to tiled render targets.		
Mip-mapped and arrayed surfaces are supported with MCS buffer layout with these alignments in the RT space: Horizontal Alignment = 256 and Vertical Alignment = 128.		
Clear is supported only on the full RT; i.e., no partial clear or overlapping clears.		
The following table describes the RT alignment:		
	Pixels	Lines
TiledY RT CL		
bpp		
32	8	4
64	4	4
128	2	4
TiledX RT CL		
bpp		
32	16	2
64	8	2
128	4	2
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.		

	Pixels	Lines
TiledY RT		
bpp		
32	128	128
64	64	128
128	32	128
TiledX RT		
bpp		
32	256	64
64	128	64
128	64	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:

	Horizontal Scale Down Factor	Vertical Scale Down Factor
MCS CL for TiledY RCC		
bpp		
32	64	64
64	32	64
128	16	64
MCS CL for TiledX RCC		
bpp		
32	128	32
64	64	32
128	32	32

Resolve rectangle must not be scaled if MCS Resolve Optimization is disabled in the Cache Mode register.

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.
- 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 bpp	Horizontal scale down factor	Vertical scale down factor
32	64	64
64	32	64
128	16	64

MSC CL for TiledX RCC bpp	Horizontal scale down factor	Vertical scale down factor
32	128	32
64	64	32
128	32	32

The pixel shader kernel requires no attributes, but must deliver a render target write message covering all pixels and all render targets desired to be resolved. The color data in these messages is ignored (the replicated color message is required).

Programming Note	
Context:	Render Target Resolve
<ul style="list-style-type: none"> Depth Test Enable, Depth Buffer Write Enable, Stencil Test Enable, Stencil Buffer Write Enable, and Alpha Test Enable must all be disabled. This render target resolve procedure is not supported on multisampled render targets. Unresolved multisampled render targets are directly supported by the sampling engine, which resolves clear values in addition to decompressing the surface. This applies to both <i>ld2dms</i> and <i>sample2dms</i> messages. 	

Media GPGPU Pipeline

This section of the PRM 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 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.
3. Set up CURBE with thread ids and common data for all thread dispatches in the thread group.
4. 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 NO.

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 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.

The MEDIA_STATE_FLUSH command is updated to specify all the resources required for the next thread group via an interface descriptor – if the resources are not available the group cannot start.

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 Memory Interface and Command Stream for the MMIO register addresses and formats.

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_OBJECT, 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.
...	
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
<ul style="list-style-type: none"> 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_CSR_BASE_ADDRESS

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_SEMAPHORE_WAIT

Media commands like MEDIA_OBJECT and MEDIA_OBJECT_WALKER are not preemptable.

Note that command preemption is not supported for MEDIA_OBJECT_* commands – 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.

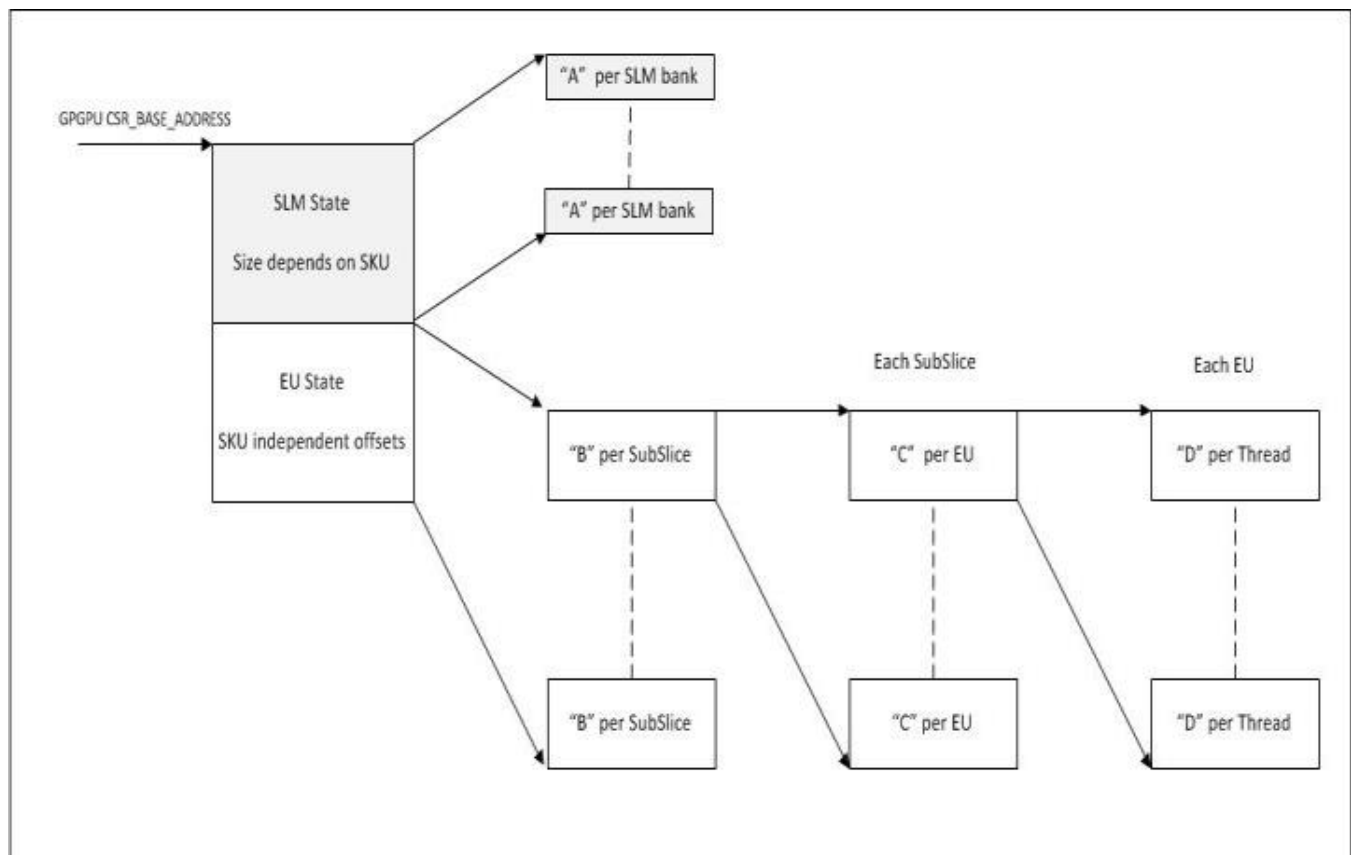
Maximum Upper Bound

Description
The maximum upper bound is set to 4MB for GT2 and GT3.

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.NumSImBanks] + 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.

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

For the BDW B0 stepping, inline payload cannot be combined with the scoreboard for media workloads.

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 PRM section on 3D pipe synchronization: [vol2a 3D Pipeline - Overview](#) > [3D Pipeline](#) > [Synchronization of the 3D Pipeline](#).

Mode of Operations

This section contains registers for GPGPU Object and GPGPU Command. It also covers GPGPU Mode.

GPGPU Thread R0 Header

The R0 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	FTTID. 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 = [0,10] indicating [2K bytes, 2M bytes] in powers of two.
	3:0	Per Thread Scratch Space. Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two is raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two.
R0.2	31	Reserved: MBZ

DWord	Bits	Description
	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
	23:15	This field is used to transfer a debug key to distinguish transactions.
	14:10	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
	29:28	Reserved: MBZ
	27:24	Shared Local Memory Index: Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4K memory block, 16 possibilities cover the entire 64K shared memory per half-slice. Format = U4
	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.

Description
The GPGPU_WALKER command is supported.
At least one form of payload (either indirect or CURBE) must be sent with the GPGPU_WALKER.

GPGPU_WALKER

GPGPU Mode

The general purpose (GPGPU) mode allows the Gen7 architecture to be used by general purpose parallel APIs:

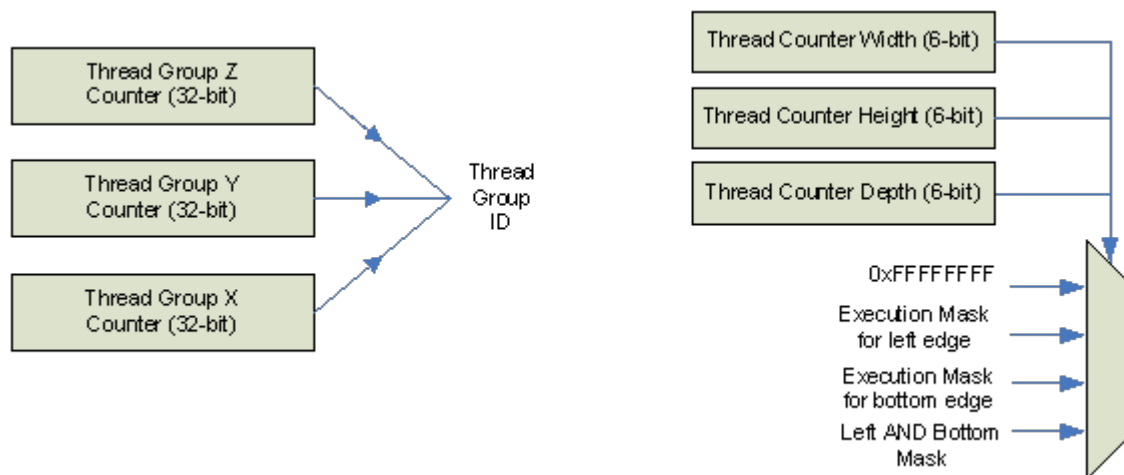
- GPGPU
- DX11 GPGPU

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.

Description
The maximum thread group size must fit into a single subslice and run in parallel, so the number of EU threads must be less than the number specified in Configurations for threads per subslice.
Some tables do not have this broken out separately. The threads per subslice = (Thread count) / (Slice Count * Samplers).

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:

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

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:

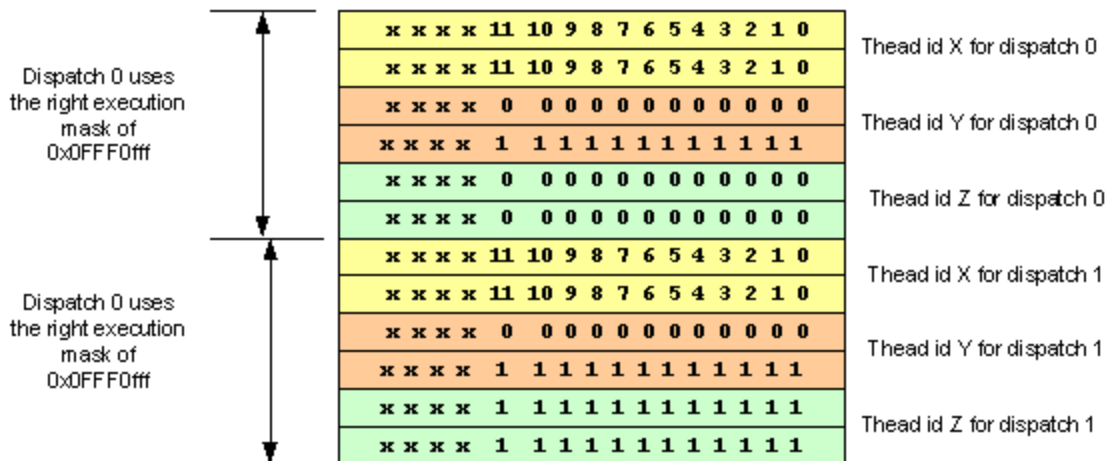
8x4	8x4	8x4	
8x4	8x4	8x4	
8x4	8x4	8x4	

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:

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

In this case the width counter can have a maximum of 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.

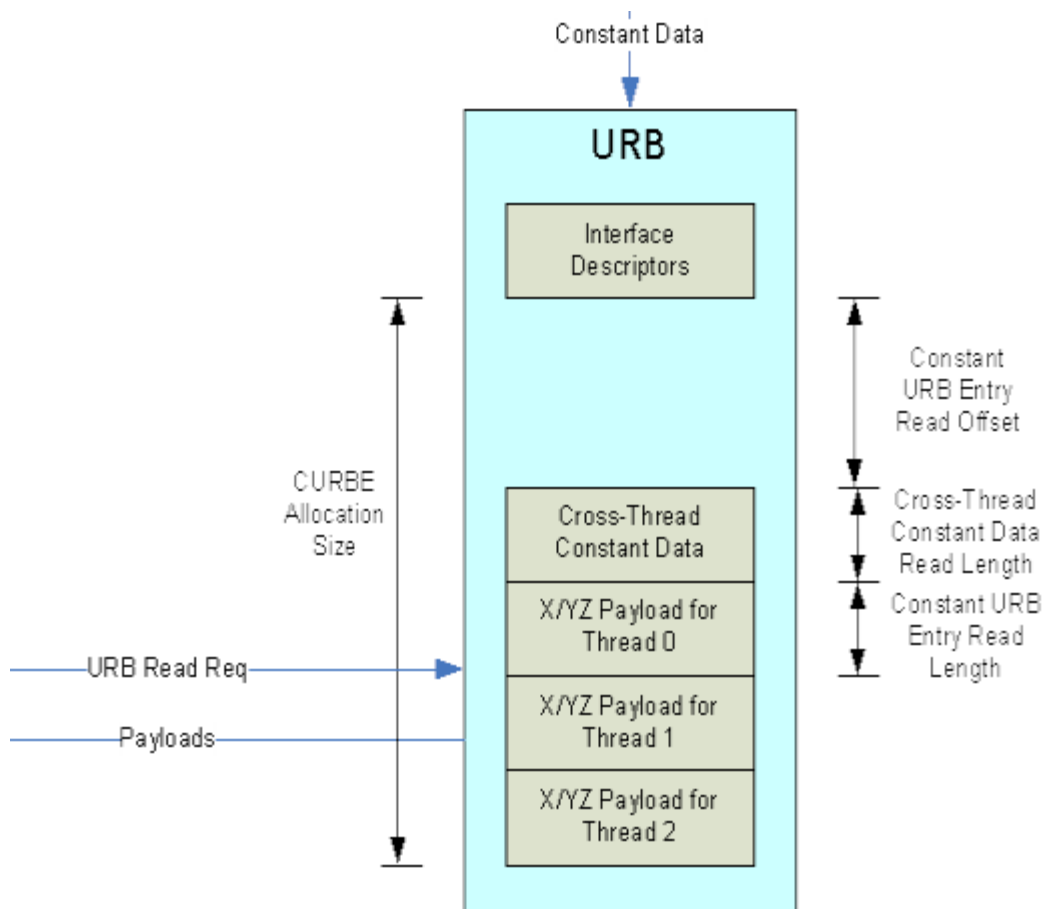
Description
<p>The first 64 URB entries are reserved for the interface description, and CURBE data is placed after the IDs. URB handles are needed for indirect data and parent/child communication; when the VFE starts up it creates up to 128 handles by partitioning the remaining URB space into evenly 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.</p> <p>MEDIA_VFE_STATE specifies the amount of CURBE space, the URB handle size and the number of URB handles. The driver must ensure that:</p> $((\text{URB_handle_size} * \text{URB_num_handle}) - \text{CURBE} - 64) \leq \text{URB_allocation_in_L3}.$

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 the 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

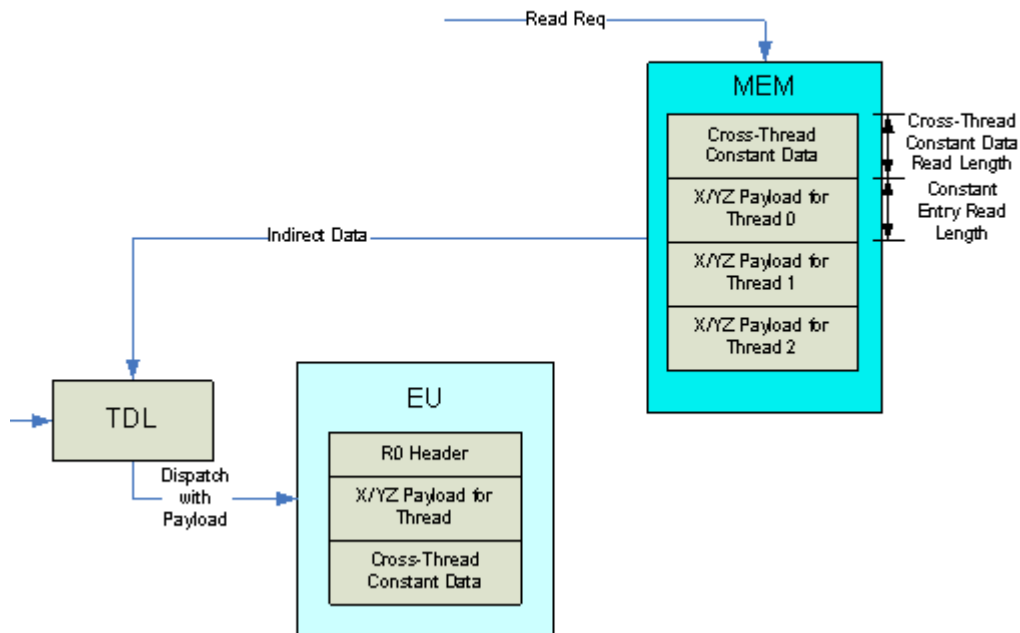


Example of CURBE command sequence:

```

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


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.

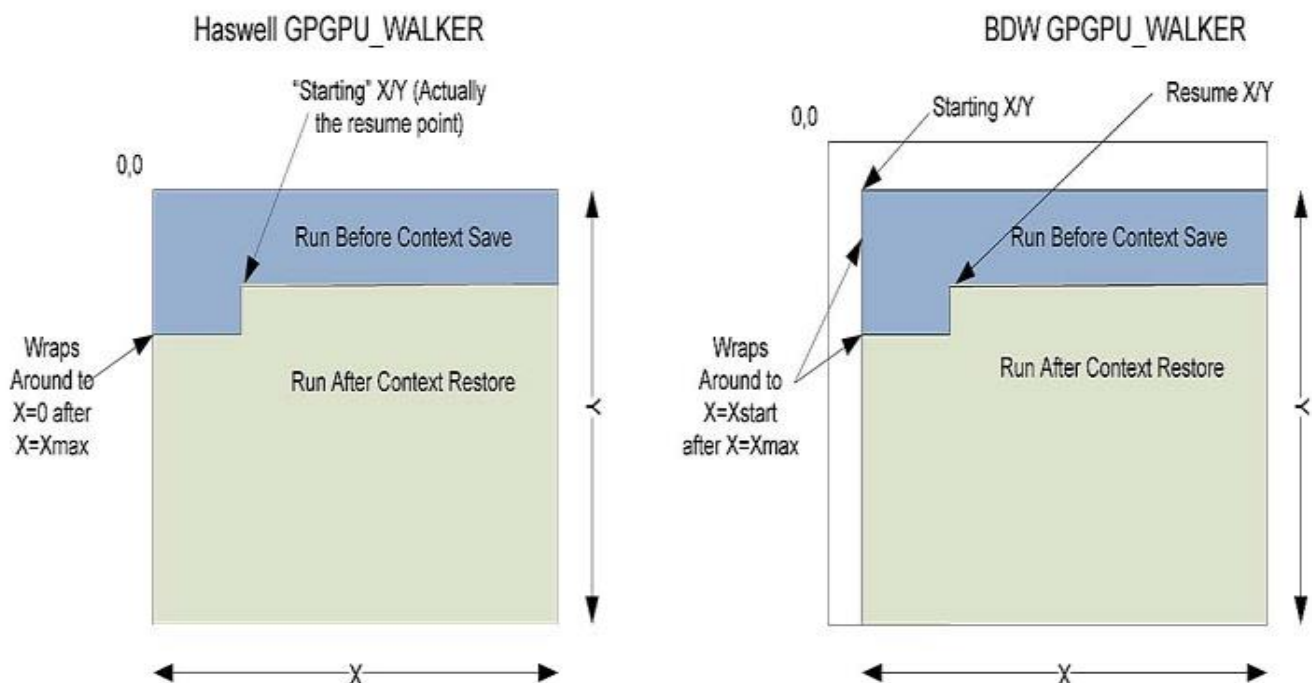
A new command called MEDIA_OBJECT_GRPID is created which is the same as the basic MEDIA_OBJECT command with the addition of a 32-bit group id and an end-of-group bit. The group id will be used to allocate barriers and SLM based on the Interface Descriptor.

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.



In BDW there is a start value that can be used to offset all the groups from 0,0,0 instead of a resume value that tell us where to resume after a context restore.

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 similar 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 course. 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 16 barriers), 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.

Generic Media

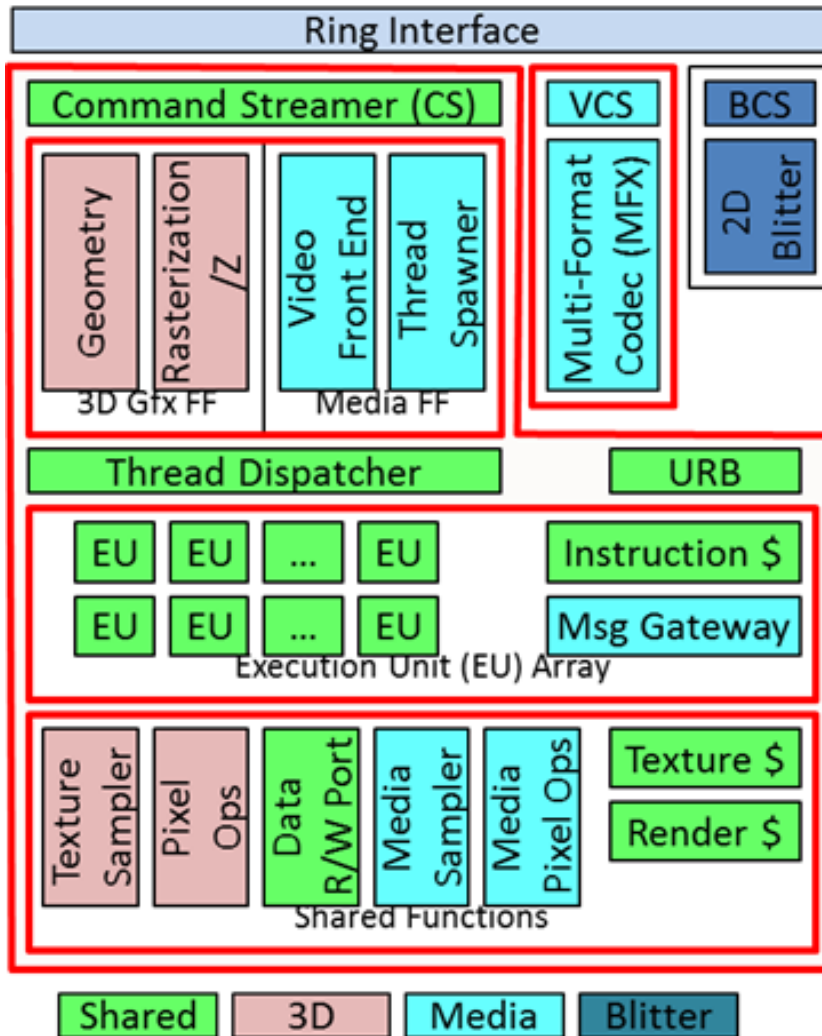
This introduction provides a brief overview of the Media product features. It includes Media functions, feature benefits, and how the features fit into graphics products as part of a whole solution.

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.

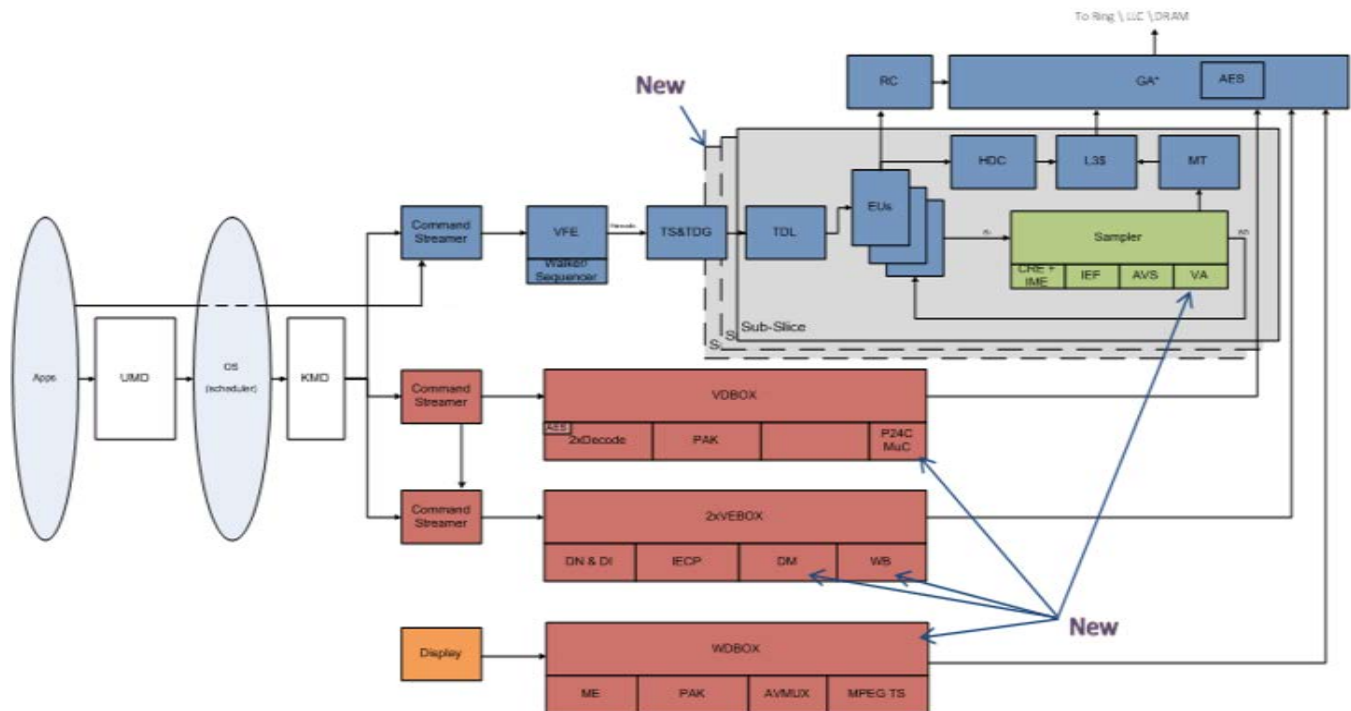


- **Fixed Function (FF) pipelines:** Provide thread generation and control.
- **3D graphics or Media FF:** Controls EU array at a given time. The EU (Execution Unit) array is shared between 3D and Media; ISA is optimized for both.
- **Shared functions:** Are accelerators to run filtered load, scatter, gather, and filter/blended store operations.
- **MFX:** Is a parallel codec engine that runs in a separate context.

Product Evolution

Block diagrams in this section describe the evolution of Media products, by project. They include definitions of the main components and how they integrate with each other.

BDW GT2 Media Pipelines



Additions/Changes for BDW:

- **2xVEBOX:** Two instances of VEBOX on the same chip.
 - **DM:** De-Mosaic (image processing used to reconstruct a full-color image from sensor output, which is usually in Bayer format).
 - **WB:** White Balance is the process of removing unrealistic color casts.
- **VA:** Computer Vision/Analytics - Accelerators for commonly used mathematical operations for computer vision functions.
- **WDBOX:** Wireless Display - Low latency, low power Widi solution.
 - **ME:** Motion Estimation.
 - **AVMUX:** Audio/Video Muxing (Multiplexing of audio and video streams into a single stream while preserving A/V synchronization).
 - **MPEG TS:** MPEG Transport Stream (as defined in ISO MPEG-2 standard).
- **Decode HW:** AVC, MPEG-2, MVC, VC-1, VP8, JPEG, HEVC and legacy formats (AVS, MPEG-4p2).
- **Encode HW:** AVC, MPEG-2, MVC, JPEG, and VP8.

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.
IQ	Inverse Quantization. It is a stage in the video decoding pipe between IS and IDCT.
IT	Inverse Integer Transform. It is the stage in AVC or VC1 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.
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

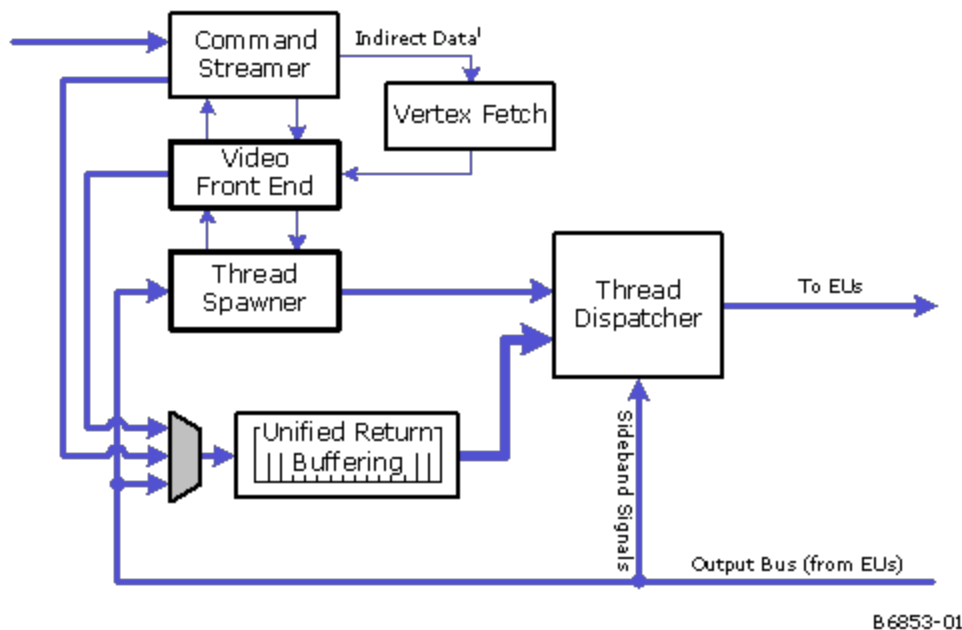
Media Pipeline Overview

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_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

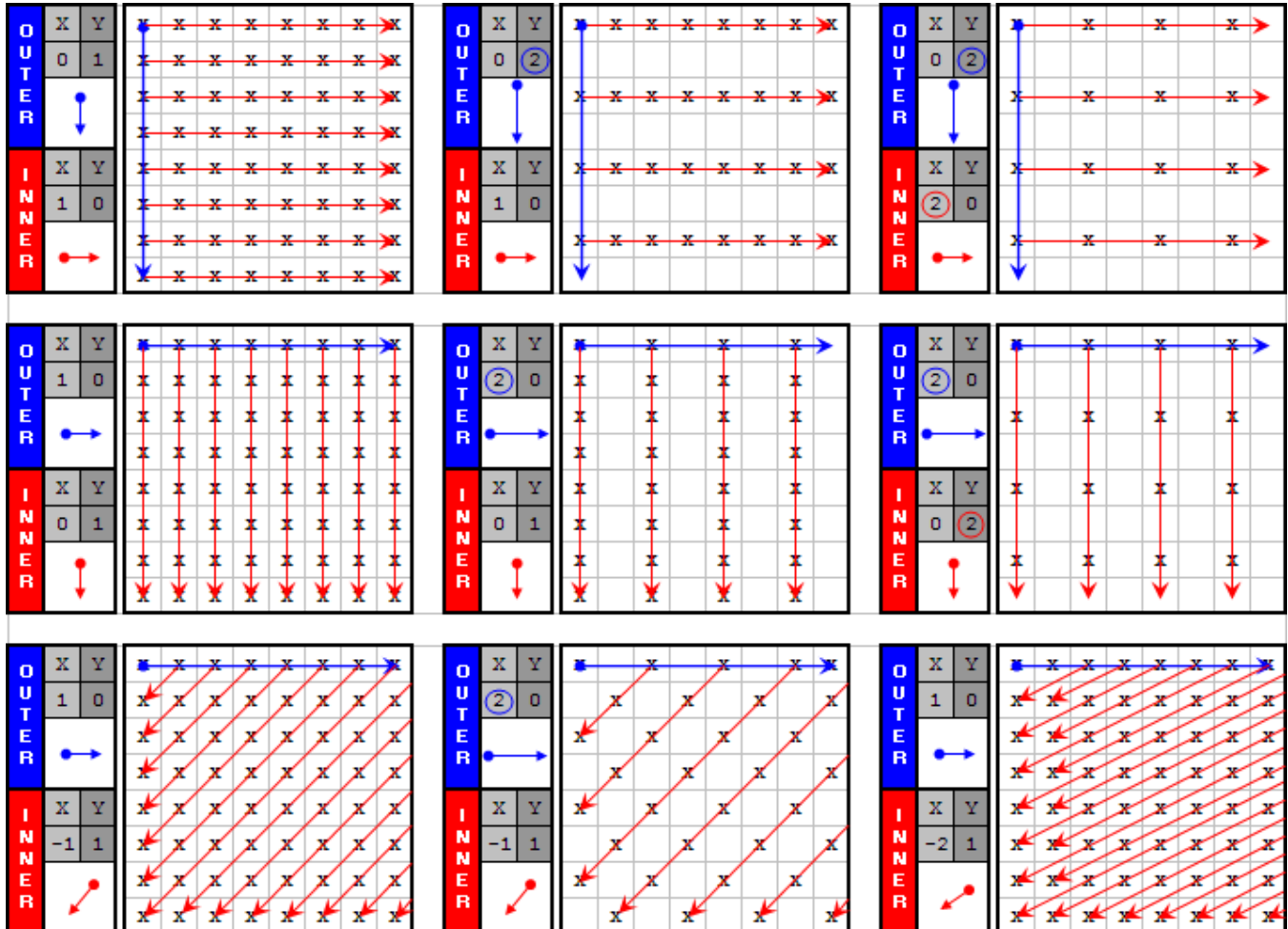
It should be emphasized that the value of what a “unit” represents is implicitly defined by the user. In other words, the walker traverses a “unit normalized space” that is not inherently bound to pixel walking. If the smallest unit of work the user wants to walk is a 4x3 block of pixels, you can program 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.

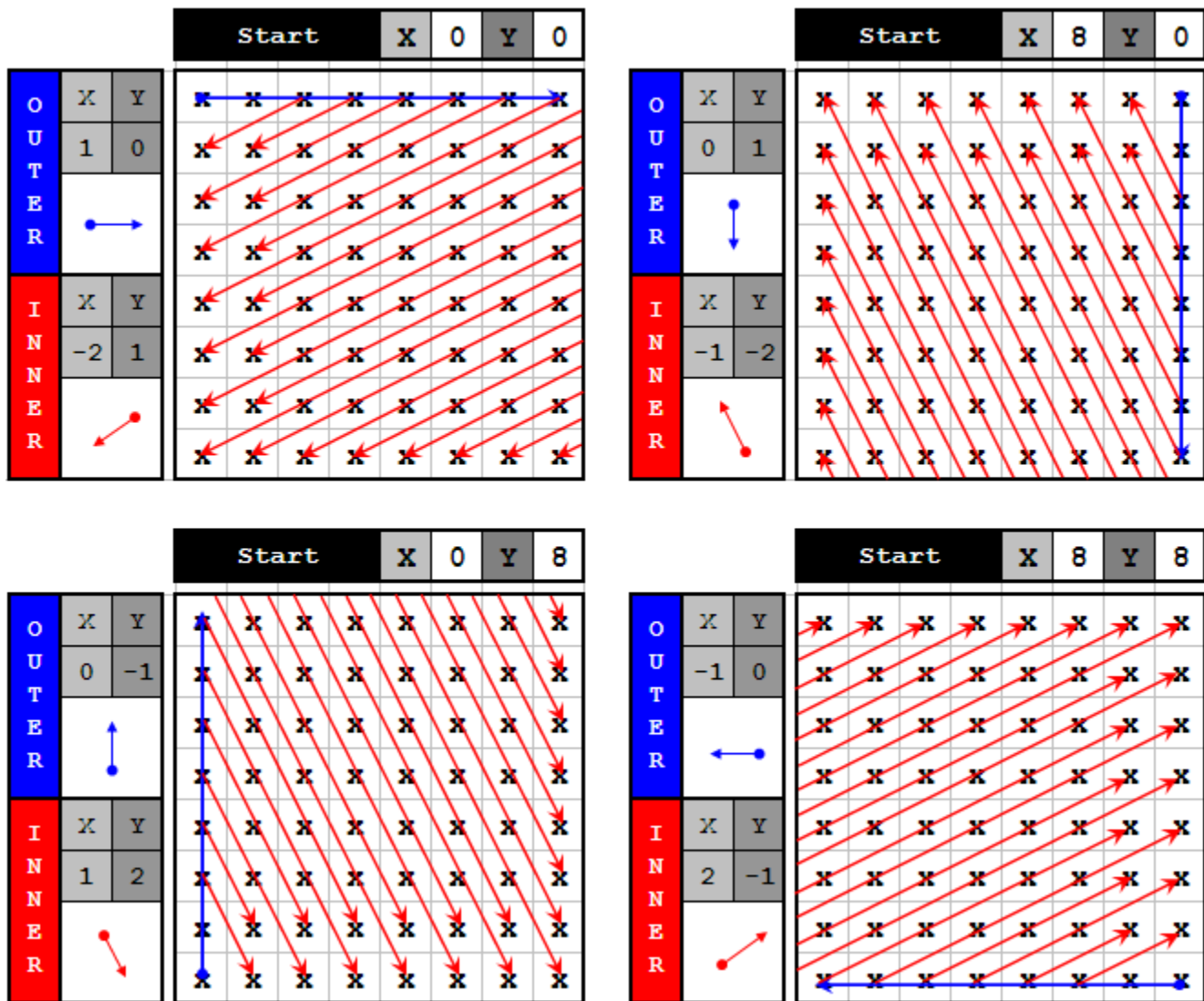
Basic Parameters for the Local Loop

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:



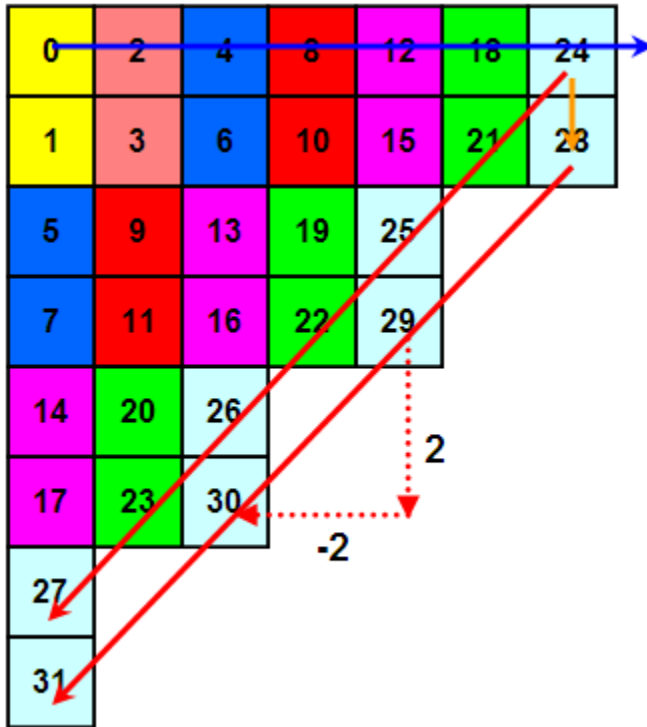
- The top row shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Rows and columns can easily be skipped by increasing the unit step vectors above one.
- The middle row the outer unit vector pointing right (+X) and the inner unit vector pointing down (+Y). Again, 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.

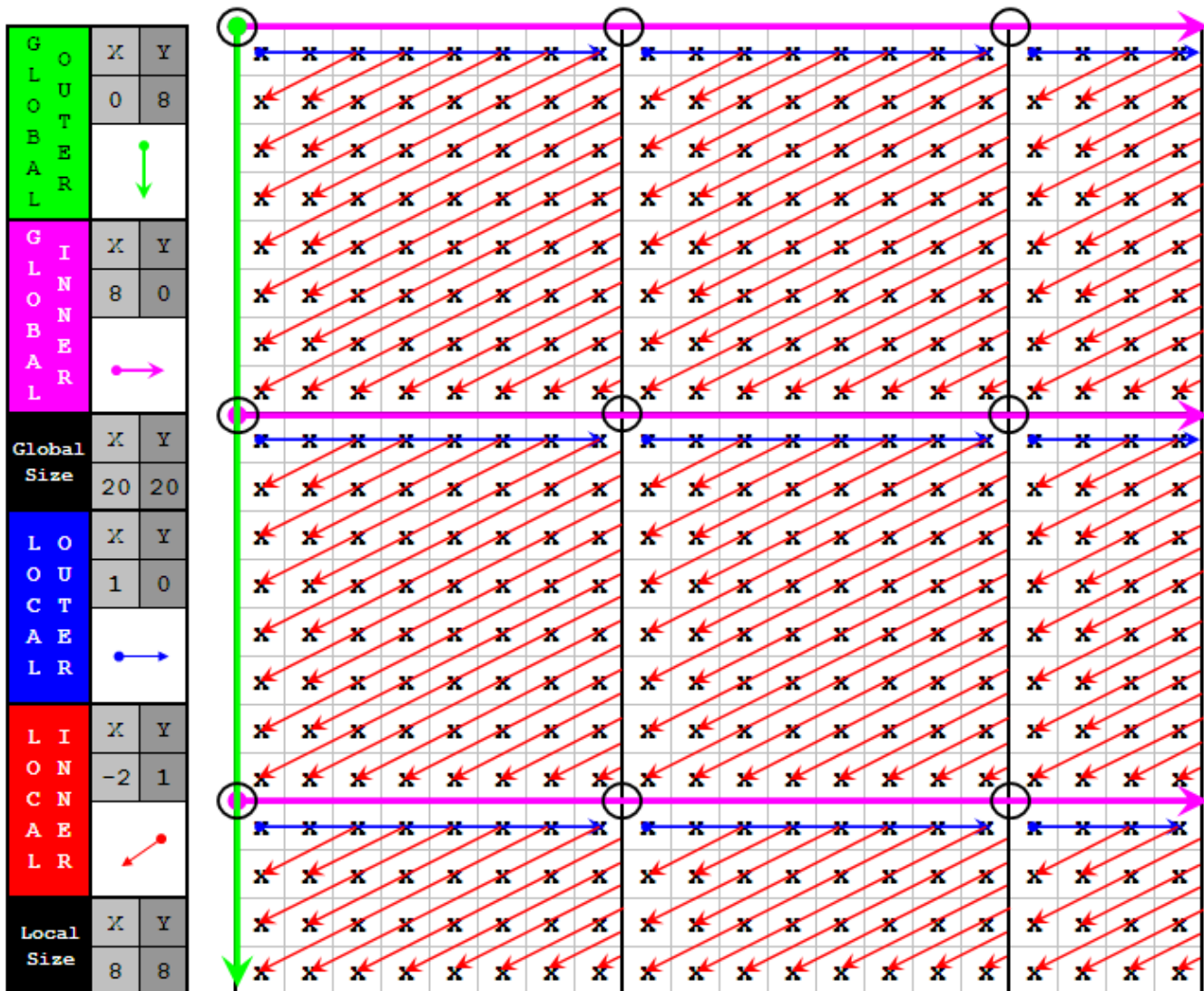


The outer loop unit step vector would be $[1, 0]$ and the inner loop unit step vector would be $[-2, 2]$. A third loop is necessary to repeat the inner loop, only shifted down a unit before restarting. Thus, a middle loop with a unit step vector of $[0, 1]$ would achieve this MBAFF pattern. Additionally, 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_Initialization();
        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:

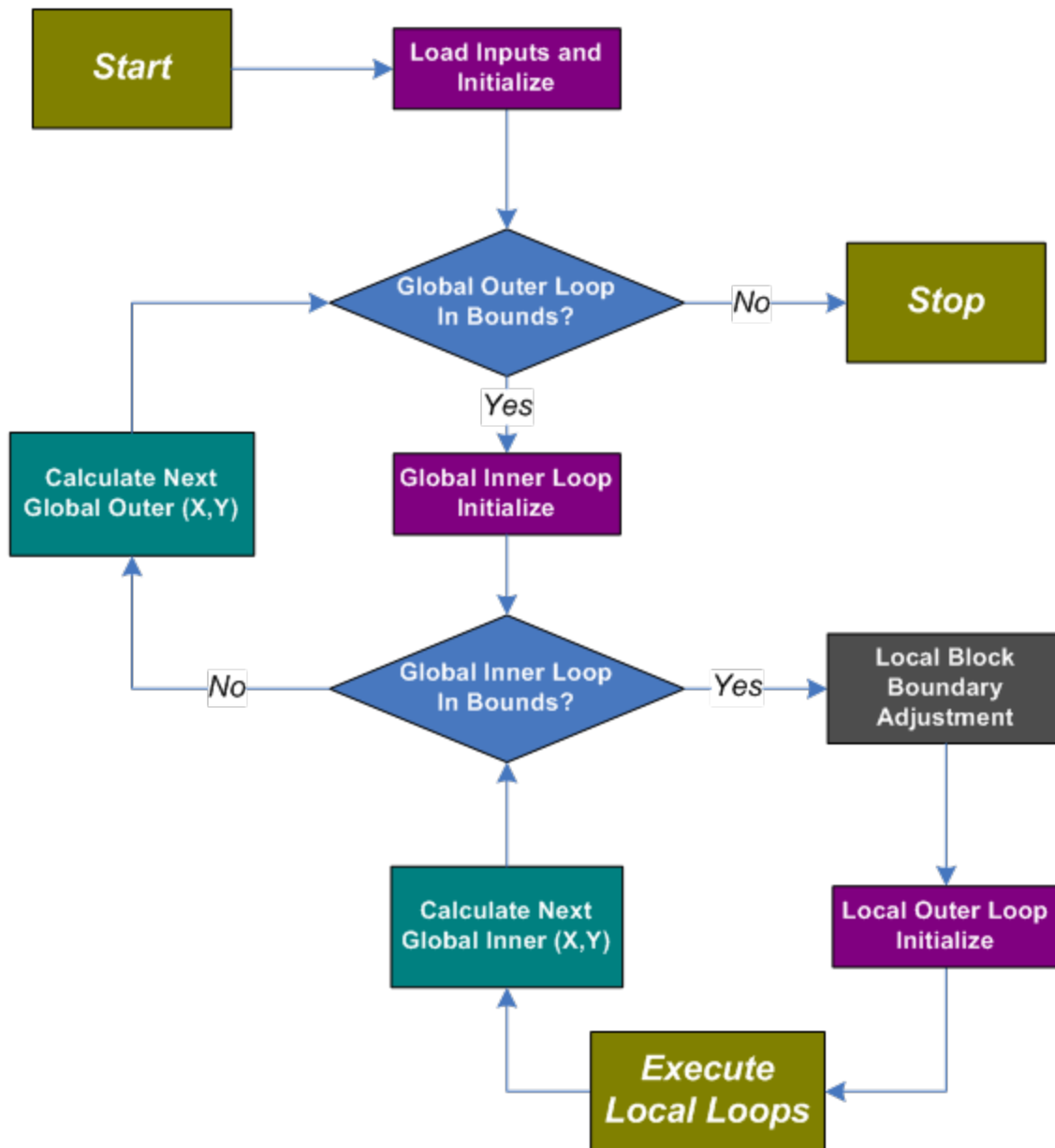
- There are 5 levels of iteration
- The highest 2 levels are called "global" and the lowest 3 levels are called "local"

- 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



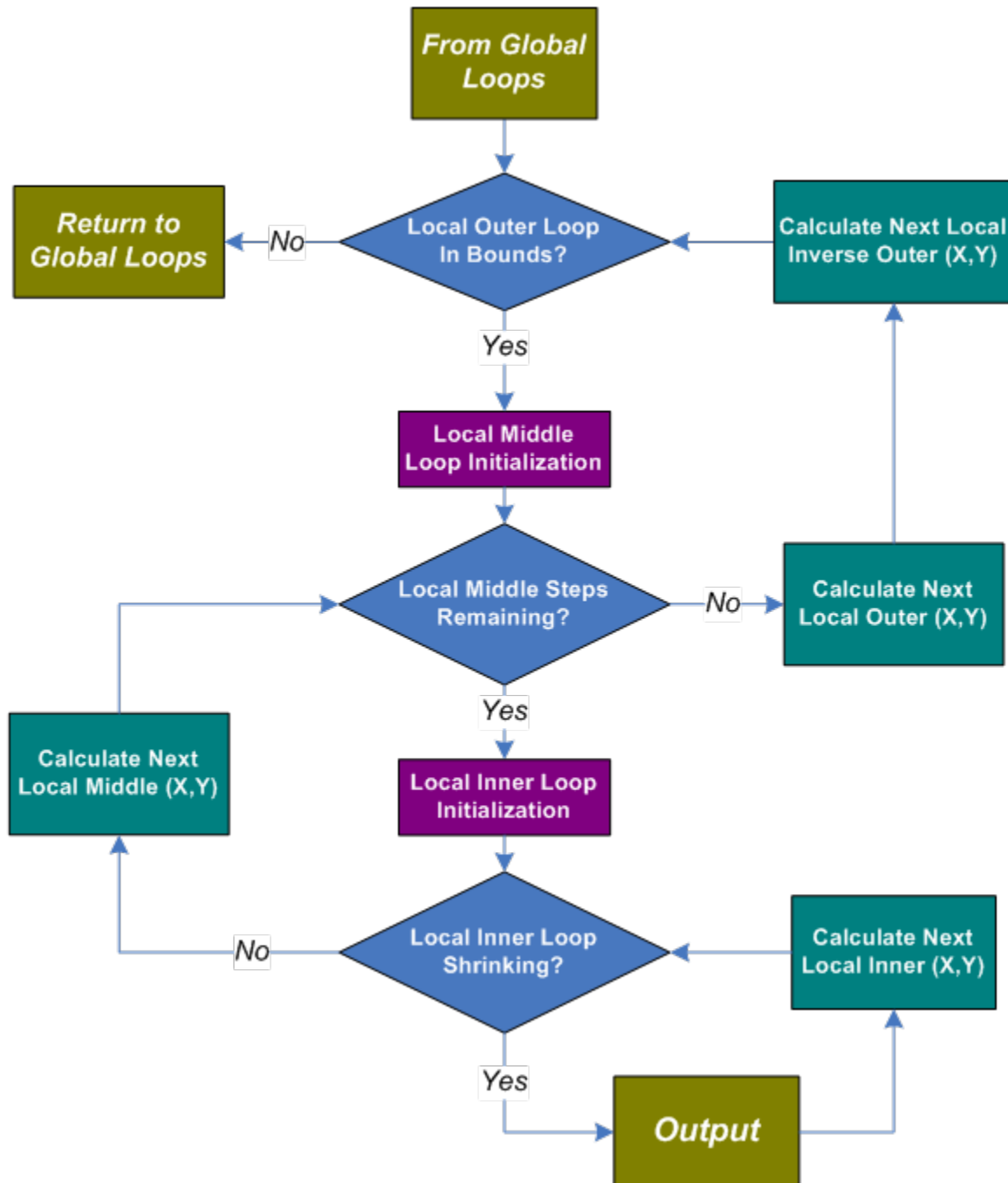
Take note of the grey box "Local Block Boundary Adjustment". This logic is necessary to adjust the local block size when the distance between the current global position to the edge of the image is less than the local resolution. Additionally, the local starting positions might be modified here as well 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.

BDW 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.

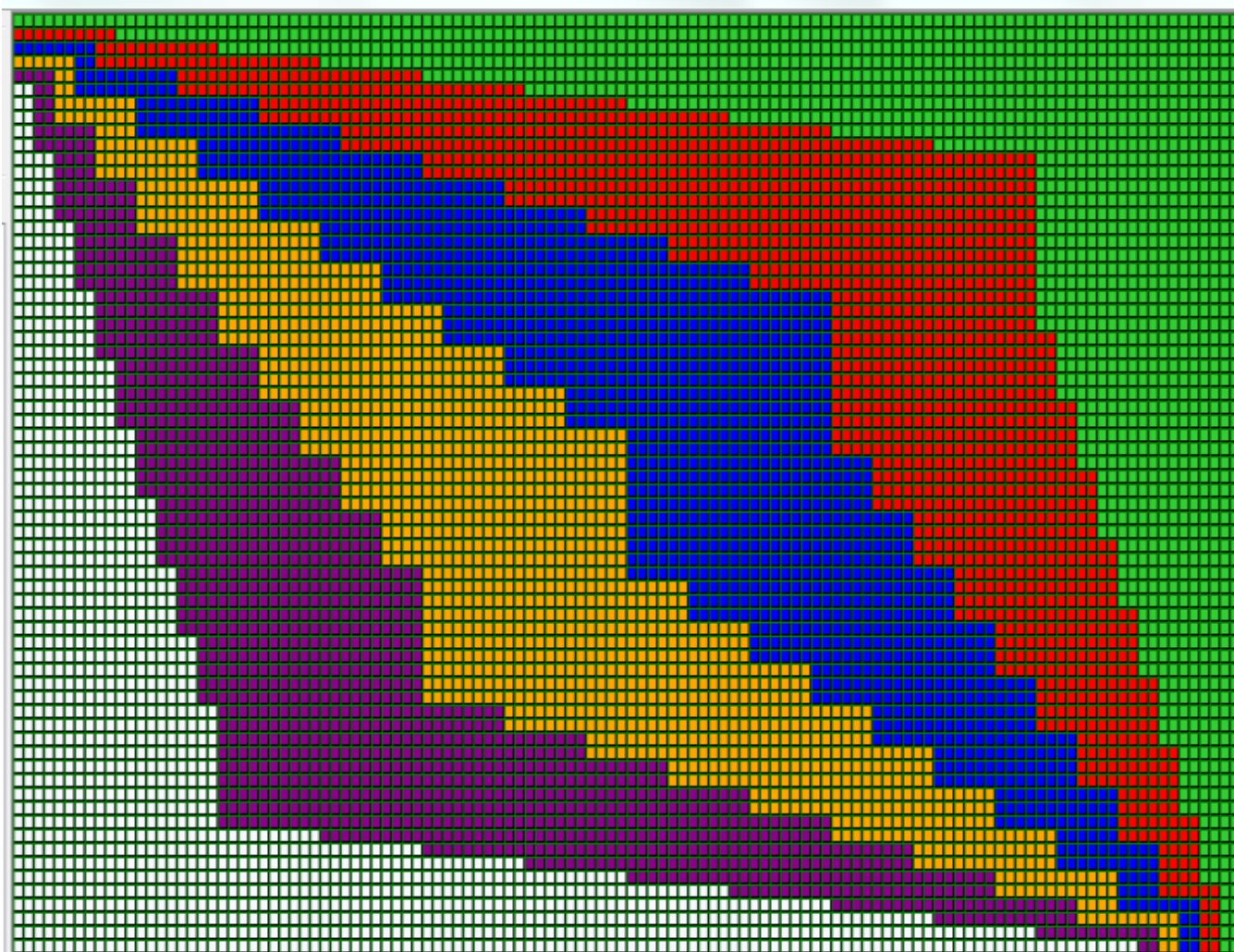
Before the MEDIA_OBJECT_WALKER command with groups there should be a MEDIA_STATE_FLUSH with a watermark bit and matching Interface Descriptor.

Flexible Dispatch of Local Loop

The local loop is automatically split between the available subslices in such a way as to keep adjacent blocks next to each other to improve cache and execution efficiency. 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 diagonal 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.

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 (ΔX , ΔY). 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 + ΔX , Scoreboard Y + ΔY), 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 ($\Delta X = 0$, $\Delta Y = 0$) 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
<ul style="list-style-type: none"> 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

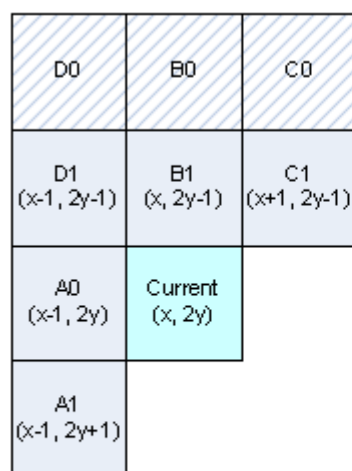
For AVC decoding, dependencies for a given macroblock may be set based on the availability of neighbor macroblocks, namely A, B, C, D and left-bottom neighbors (left-bottom only if MbAff = 1), as well as the current macroblock's address, MbAff flag and FieldMbFlag. For a macroblock in a progressive frame 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.

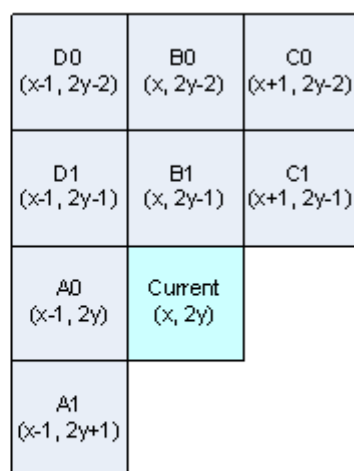
Neighbor addresses of a macroblock in a progressive frame picture (MbAff = 0) or a field picture with up to 4 dependencies

D (x-1, y-1)	B (x, y+1)	C (x+1, y-1)
A (x-1, y)	Current (x, y)	

Neighbor addresses of the first macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies

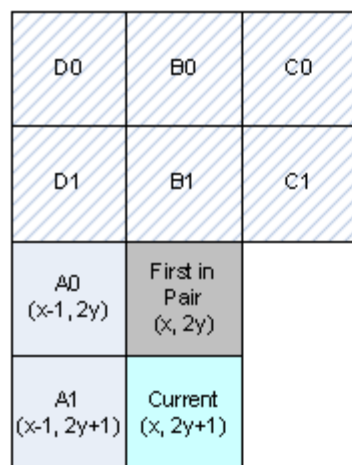


(a) Neighbors for the first macroblock in a 'frame' MbAff pair

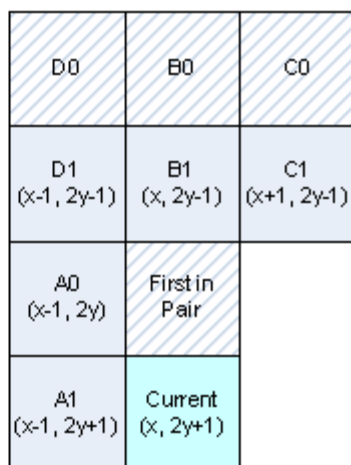


(b) Neighbors for the first macroblock in a 'field' MbAff pair

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	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Progressive or Field picture
1	0	0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1 st Frame MbAff macroblock
1	0	1	<input type="checkbox"/>	na	0	na	<input type="checkbox"/>	2 nd Frame MbAff macroblock
1	1	0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1 st Field MbAff macroblock
1	1	1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	2 nd 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
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.

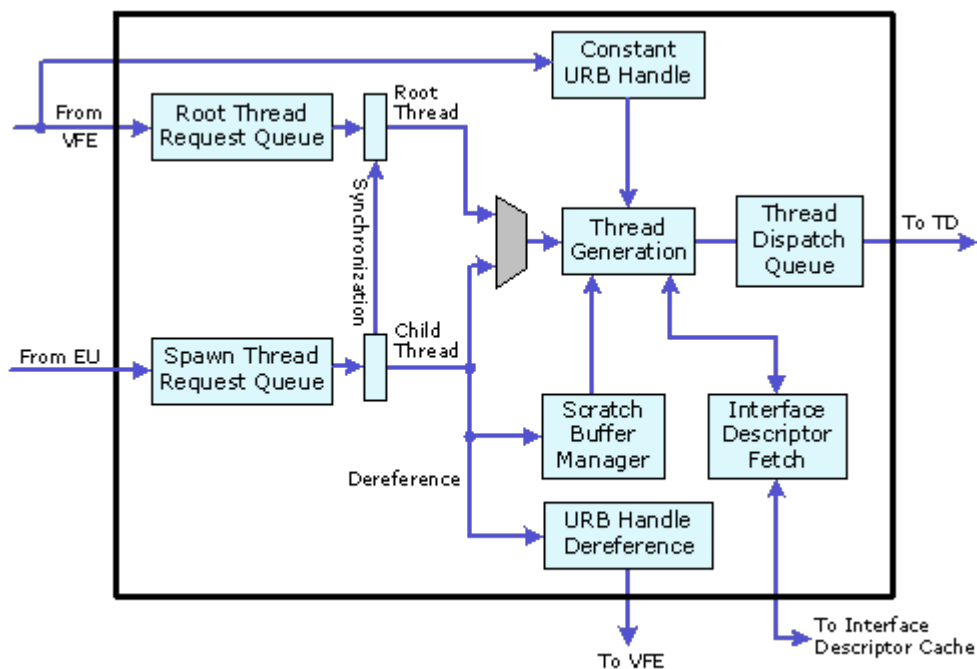
In modes with child threads, a root thread may persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. The Software designer should partition the workload to restrict the 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



B6857-01

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.

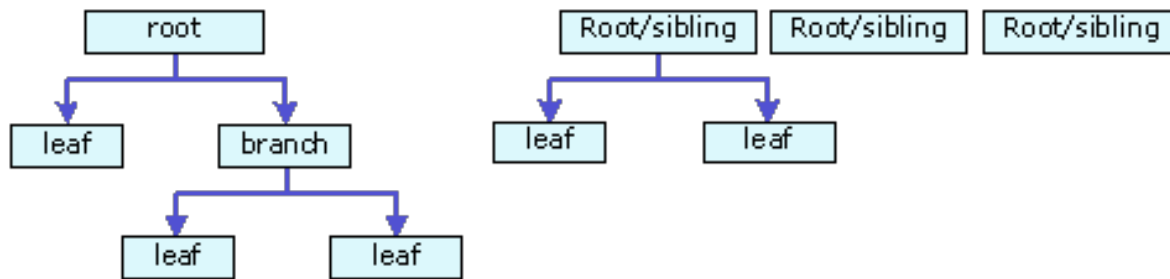
TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

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.

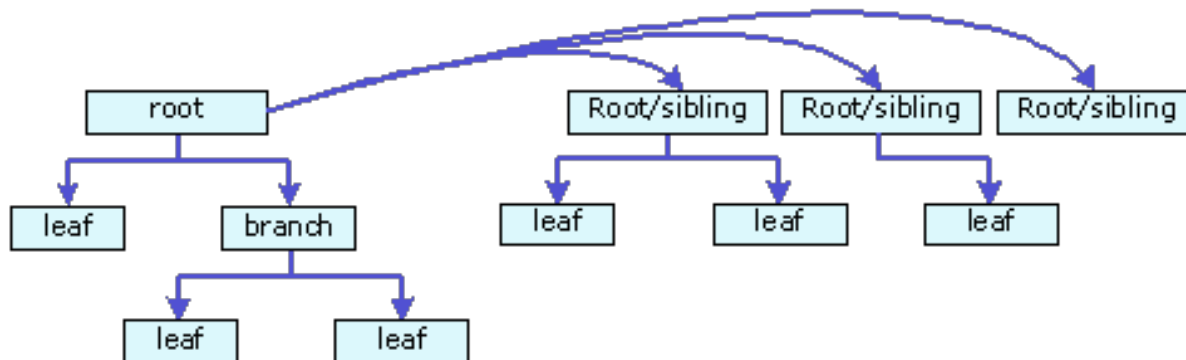
There is another case (as shown in *Multiple Simultaneous Roots*) where multiple trees may be connected. If a root is a synchronized root thread, it may be dependent on a preceding sibling root thread or on a child thread.

Examples of thread relationship



B 6858-01

A example of thread relationship with root sibling dependency



B 6859-01

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.

$$\text{Max_Outstanding_Child_Threads} = (\text{Thread Slot Number} - 1) + (\text{TS Child Queue Depth} - 1)$$

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.

$$\text{Max_Outstanding_Child_Threads} = (\text{Thread Slot Number} - 1) + (\text{TS Child Queue Depth} - 1) - (\text{Max Root Threads} - 1)$$

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.

Media Messages

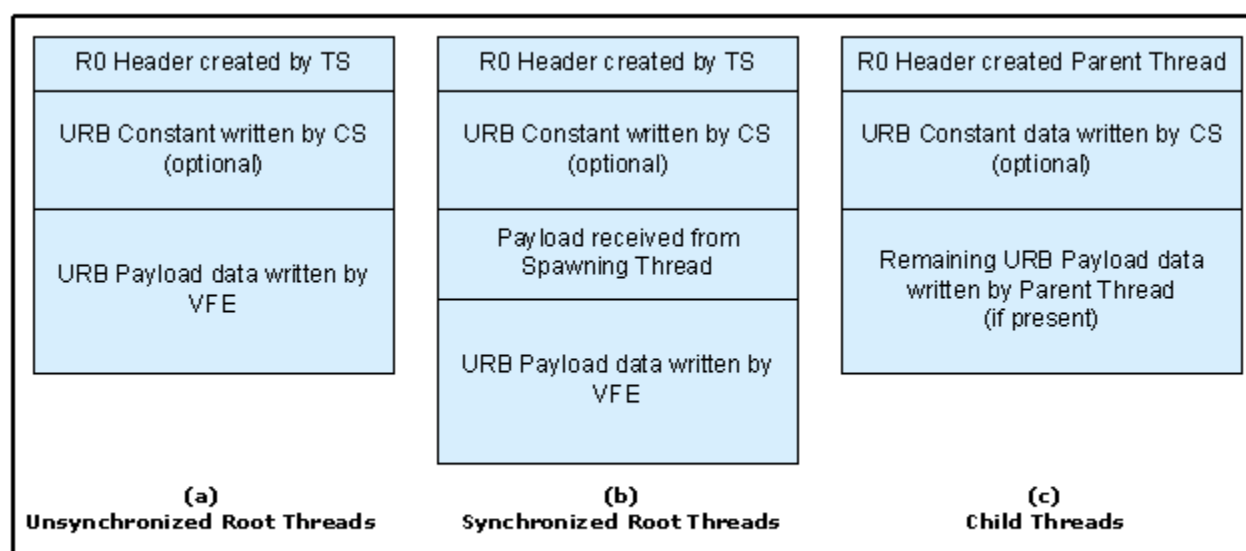
All message formats are given in terms of dwords (32 bits) using the following conventions:

- Dispatch Messages: **Rp.d**
- SEND Instruction Messages: **Mp.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



B6863-01

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]
	9	Reserved: MBZ
	8:0	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. Format = U9
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
	26:25	Reserved: MBZ
	24:16	Scoreboard Y This field provides the Y term of the scoreboard value of the current thread. Format = U9
	15:11	Reserved: MBZ
	10:9	Reserved: MBZ
	8:0	Scoreboard X This field provides the X term of the scoreboard value of the current thread. Format = U9

DWord	Bits	Description
R0.0	31:24	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
	23:16	Reserved: MBZ
	15:0	URB Handle. This is the URB handle indicating the URB space for use by the root thread and its children.

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

DWord	Bit	Description
	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. 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 <i>this thread's</i> 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	<p>Group ID LSB. This is the LSBs of the Group ID. For MEDIA_OBJECT_GRPID threads this is the entire group id. For MEDIA_OBJECT_WALKER threads the interpretation depends on the Group ID Loop Select:</p> <p>0: No groups, field is 0.</p> <p>1: cat(InnerGlobalCnt[6:0], OuterLocalCnt[9:0], MidLocalCnt[4:0], InnerLocalCnt[9:0]); rest of group id is in R0.2.</p> <p>2: cat(OuterGlobalCnt[6:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0], MidLocalCnt[4:0]); rest of group id is in R0.2.</p> <p>3: cat(2'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0]).</p> <p>4: cat(12'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0]).</p> <p>5: cat(22'b0, OuterGlobalCnt[9:0]).</p>
R0.5	31:10	<p>Scratch Space Pointer. Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled.</p> <p>Format = GeneralStateOffset[31:10]</p>
	9	Reserved: MBZ
	8:0	<p>FTID. This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.</p>
R0.4	31:5	<p>Binding Table Pointer. Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address.</p> <p>Format = SurfaceStateOffset[31:5]</p>
	4:0	Reserved: MBZ
R0.3	31:5	<p>Sampler State Pointer. Specifies the 32-byte aligned pointer to the sampler state table.</p> <p>Format = GeneralStateOffset[31:5]</p>
	4	Reserved: MBZ
	3:0	<p>Per Thread Scratch Space. Specifies the amount of scratch space, in 1K-byte quantities, allowed to be used by this thread. The value specifies the power to which two is raised to determine the amount of scratch space.</p> <p>Format = U4</p> <p>Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two</p>
R0.2	Reserved	
RR0.1	31:27	Reserved: MBZ
	26:25	Reserved: MBZ

DWord	Bits	Description
	24:16	Scoreboard Y. This field provides the Y term of the scoreboard value of the current thread. Format = U9
	15:11	Reserved: MBZ
	10:9	Reserved: MBZ
	8:0	Scoreboard X. This field provides the X term of the scoreboard value of the current thread. Format = U9
R0.0	31:30	Reserved: MBZ
	29:28	Reserved: MBZ
	27:24	SLM ID. This field indicates the index to the SLM starting offset associated with this kernel. Format: U4
	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	31:8	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. 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	31:28	Ignored.
	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.
	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	31:28	Ignored.

DWord	Bits	Description
	27:24	Shared Local Memory Index. Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4K memory block, 16 possibilities cover the entire 64K shared memory per half-slice. Format = U4
	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.

L3 Cache and URB

This section describes the GFX L3 Cache, which is a large storage 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 section discusses GFX L3 cache for BDW. The included topics are:

- Overview
- L3 Coherency
- L3 Allocation & Programming
- Shared Local Memory (SLM)
- L3 Cache Error Protection
- L3 Registers

Overview

GFX L3 cache is a large storage which backs up various L2/L1 caches on many clients. It provides a simple way based partitioning option for each or 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 (shared local memory) for EUs/ROWS.

In order to provide the bandwidth needed L3 has been separated into 4x192KB structures which can be accessed concurrently. A 2x clocking is introduced to further enhance the bandwidth and cover the limitations of SRAM design.

- Formed as 4 (2 for GT1, 8 for GT3) individual banks each with 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 are
 - 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)

Gen8 L3 controller added an optimization where back 2 back access to the same address are performed full bandwidth rather than partial serialization that was on previous generations. This optimization eliminates the need for software work around of distributing or moving of high accesses buffers within a short temporal period.

L3 Cache Configuration

- 4x192KB cache, 96 logical ways
 - Upto 64 ways tagged for L3\$, remaining is treated as memory.
 - Shared local memory capable (up to 256KB per slice, i.e. GT2)
- 64B Cacheline with a portion capable of highly banked memory (with 16x4B)

- Interface 4x64B to SQDB for the fill/write path, 4x64B Read/Evict path to SQDB. Additional 4x64B read and 4x64B write capability for SLM
- Data Array built via 6T cells
 - Data protection via parity/ECC
- TAG/LRU/STATE (using gen-ram via RLS flows)
 - 39/48-bit addressing support in TAG
 - 5 bit state
 - Intel pseudo-LRU implementation for selecting the line to be replaced
- Repetition rates for each operation
 - All operations – 1 every 2x clock
 - With b2b restriction for same type of accesses (i.e. read to read or write to write)

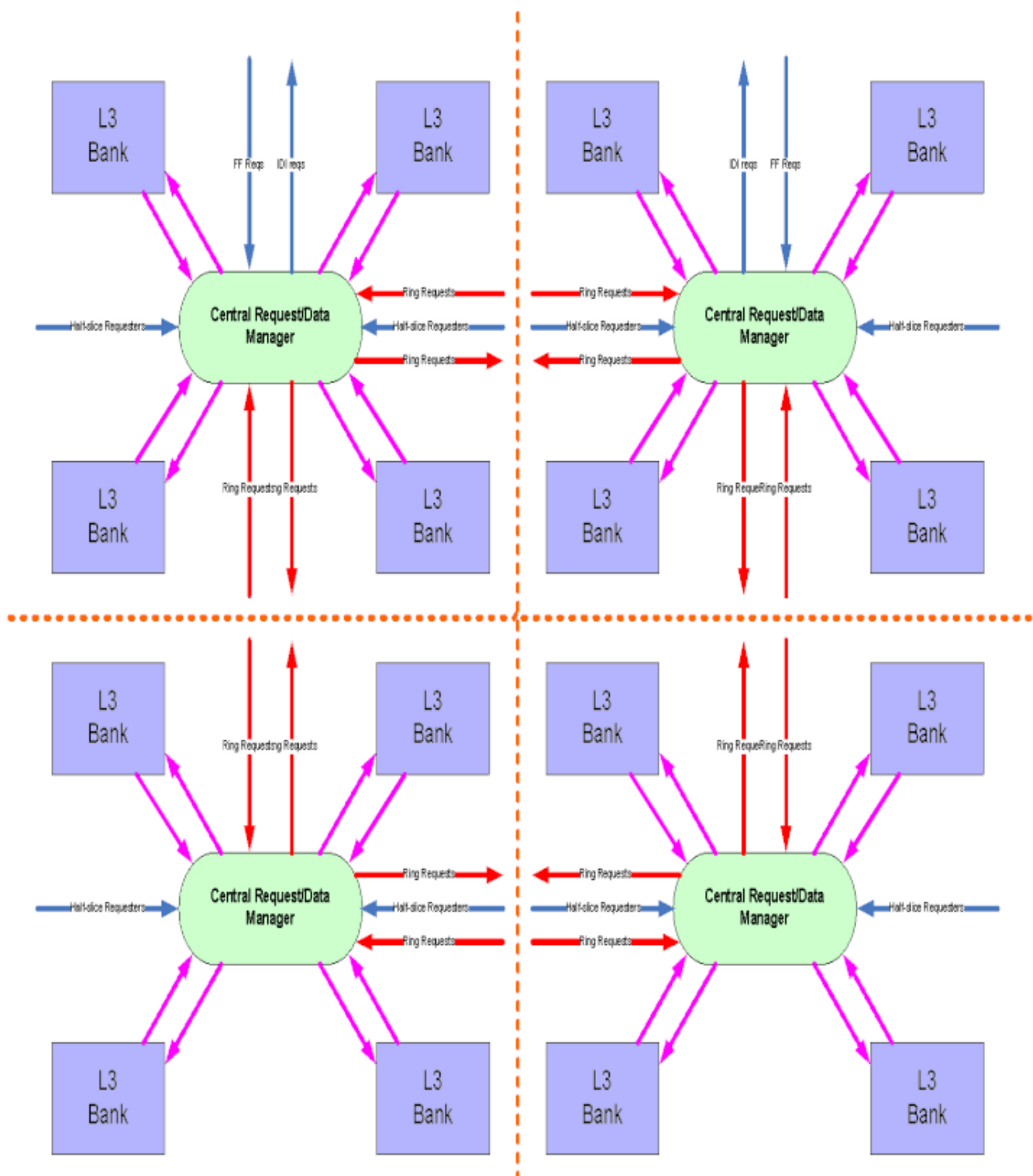
Block(s) Overview

L3 is formed via 4 (2 for GT1) logical banks that are very similar to each other in formation. The major blocks in each logical banks

- 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

Rest of the document will go through the details of these blocks and provide details of their logical operation. In addition there will be specific sections that will go through the requirements for coherency.

Vision is to build a compute scalable cache where with each additional compute both the size and bandwidth is 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 for 8th Generation

Non-SLM Mode			
	GT3 (1.5MB)	GT2 (768KB)	GT1 (384KB)
URB	512KB-768KB	256KB-384KB	128KB-192KB
Shared local Mem	0KB	0KB	0KB
L3\$	768KB-1MB	384KB-512KB	192KB-256KB

SLM Mode			
	GT3	GT2	GT1
URB	256KB	128KB	64KB
Shared local Mem	512KB	256KB	128KB
L3\$	768KB	384KB	192KB

ECC Protection

8th generation L3 has added ECC (SECCDED) protection on a 32B level to provide better data protection. The algorithm is an industry standard. Double bit errors are reported to s/w.

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)
- Tessalator (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 the data port where both could be addressed L3 or URB. The destination field is part of the request and could be set to redirect the request to:

1. L3 cache
2. URB
3. State Arbiter
4. 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.

Memory Object Control State on Cacheability

See Memory Interface descriptions for MOCS definition.

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. SW 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 the following client pools:

- **Shared Local Memory:** When enabled its size is always fixed to 128KB (64KB for GT1)
- **URB:** Local memory space, provides a flexible allocation on per 8KB 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

Each of the L3 way allocations are managed via pLRU, hence 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 represent the most likely cases for different operation modes.

For GT1, hardware will retain 2 of the L3 banks hence all following allocations will be reduced half the size.

The URB size programming given in this section shows L3 capability but does not necessarily dictate the SW programming requirements. The programming should be based on what SW chooses in 3D command for 3DState_URB_* selection. Once the URB size is determined, the remaining area can be given to L3\$.

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 will avoid RO cache invalidation colliding with following LRI.)

After GFX soft reset, it is possible that Read-Only lines in L3 may not get cleared if L3 programming was split between read/write and read-only space. It is required for GFX driver to issue a Pipe_control with RO invalidation to clear these lines post reset before sending a new context.

Non-SLM Mode Allocation

Normal L3/URB mode (non-SLM mode), uses all 4 banks of L3 equally to distribute cycles. The following allocation is a suggested programming model. Note all numbers below are given in KBytes.

The following settings are given for GT2.

GT1 has the half of the allocation listed below.

GT3 has the 2x of the allocation listed below.

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

Normal Banked - No SLM									
3	0	256	0	0	512	0	0	0	768
4	0	256	512	0	0	0	0	0	768

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 address does not need to be flushed between configuration changes.

SLM Mode Allocation

With the existence of Shared Local Memory, a 64KB chunk from each of the 3 L3 banks will be reserved for SLM usage. The remaining cache space is divided between the remaining clients.

Note: Even though there are 4 available banks per slice which leads to reservation of 4x64KB SRAM space for SLM, only 3 of these banks usable due to SLM pool capacity being limited to 3 subslices.

Shared Local Memory Mode - highly banked Memory (per GT2 slice)								
	SLM	URB	Rest	DC	RO(I/S/C/T)	I/S	C	T
1	192	128	384	0	0	0	0	0
2	192	128	0	128	256	0	0	0
3	192	128	0	256	128	0	0	0

URB is limited to 1008KB due to programming restrictions.

Unlike previous generation logic is simplified via combining RO clients under one pool, given most clients have already their L1 and L2 caches. Bundling them greatly reduces the timing paths and complexity in trying to determine which client belongs to which way.

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 through 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.

L3 Cache Error Protection

L3 cache error protection is covered via ECC (secced) and DPF.

Both global memory and URB accesses 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 stall condition is required, GFX driver is required to initialize the URB prior to execution.

Shared local memory (SLM) accesses are protected via DPF (dynamic parity detection).

BDW L3 cache and URB has 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 is 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 "h/w lock-up" mode (enabled via MMIO) to protect the contents. Software has to make a choice:

1. For the cases where workload does not require special double-bit protection, there is no need to enable the "h/w lock-up" mode.
2. For the cases where workload demands an accurate operation and relies on h/w detected double-bit errors to lead to h/w lock-up, s/w will have to initialize the contents of URB or SLM (based on workload usage model).

Dynamic Parity Feature for GFX L3 Cache

This document is meant to outline and describe dynamic parity detection function in BDW (gen8) graphics L3 cache.

Feature Definition

The concept of DPF is to provide a run-time protection for graphics L3 cache via parity detection and redundant rows. Parity errors are considered to be an extremely rare occurrence, but this mechanism provides a means to address them should they occur.

In order to mitigate error detection needs of GFX L3 cache, parity detection capability has been added to each subbank of L3 (i.e. 4KB with 32bit I/O). A subbank is defined as a 16KB entity of the total 512KB cache. This would translate into 16 independent subbanks which all have 2 independent redundant rows for a total of 32 replacements per slice. Redundant row can be activated to replace a BAD row via writing the address of the bad row into hardware registers.

DPF is meant to detect and fix persistent bit failures where same susceptible bits fail multiple times. It was not designed to deal with random errors (i.e. SER).

On gen8, only the SLM banks require DPF which is a portion of the L3 bank.

Hardware and Software Flows

Parity Generation & Detection

GFX L3 cache will generate 1-bit parity as the data gets stored in the cache. 1-bit parity will be written along with data for future verification. As same content is accessed later in time, h/w re-calculates the parity based on read content and compares with stored parity value.

Once a mismatch is detected h/w generates a parity interrupt to GFX driver for service. In addition h/w hangs on the parity error preventing the exposure of the data to outside.

Along with the interrupt h/w will update a set of registers to indicate which bank/subbank/row has the error detected. GFX driver will need to go thru all L3 banks Error Status register looking for which bank created the error LOG (4 for GT2, 8 for GT3).

Correction Using Parity Error data and Redundant Rows

Each subbank contains 2 redundant rows which would be used to replace the bad rows that are getting reported by the h/w. GFX driver upon getting the interrupt for parity will access the report registers and record the bank/subbank/row information. This information will be stored in a permanent (non-volatile memory) location of the GFX driver for future use.

GFX driver will reset the render engine (i.e. render specific reset) to prevent the potential bad data to be permanently stored in the system. Such case has various boundary cases for s/w where the application that has the bad data should be prevented to complete or switch out.

Post reset, GFX driver will program the BAD row information into corresponding bank/subbank registers and resume operation. GFX L3 hardware will use the bad row information from h/w registers to replace the address mapping of bad row with an already existing redundant row.

Note that redundant rows themselves may have parity errors and they will be reported as the row they are replacing. S/W should recognize the use of redundant row and manage the skipping. Typically if redundant rows are faulting, that will create an unrecoverable part.

Number of Corrections

Given DPF is designed to deal with persistent errors, GFX s/w needs to be able to identify which subbank rows are producing the most number of errors. Hence the driver keeps a log of worst (some number > 2) violators (rows) of a given subbank and only fixes TOP 2 bad rows. This is done via a keeping the count of failures for each row and comparing to find the rows that have multiple failures. Given random failures are extremely rare, persistently bad rows will bubble up to the TOP.

S/W's tracking is saved in non-volatile memory, so it can keep track of bad rows across reboot/reset.

Summary

Basic Algorithm:

- Each subbank has two redundant rows. (HW design)
- SW keeps track of each subbank's worst rows (i.e. those with highest error count).
- SW's tracking's is saved to the non-volatile memory, so it persists across reboot/etc.
- At all times, the two redundant rows are used to replace the two worst offenders.
- SW always forces TDR on L3 errors.

Strengths:

- Errors dealt with immediately. (No need to determine VCC min vs. SER.)
- Redundant rows put to best use, as best as we can predict.
- SER's naturally fall-out (since low error count).
- Clean implementation.

Sub-banks with more than two persistent parity error rows

This is the case where part is bad where h/w keeps reporting more than 2 rows per subbank creating bad rows. For this case GFX driver will keep replacing the rows and rotate the 2 bad rows information while causing TDRs. There is a good chance continuous number of TDRs will lead to BSOD, but it will prevent silent data corruptions that may go without user noticing.

Interrupt Enabling

Bit5 has been selected for L3 parity interrupts. GFX driver will have to enable interrupt paths from render command streamer (x20a8) as well as rest of the interrupt structures around GTISR (x44300), GTIMR (x44304), GTIIR (x44308), GTIER (x4430C).

Clearing the Error Reporting Registers

Clearing error status registers around L3 cache can be done via writing to L3CD Error Status Register bit[13] with a logical value of "1". GFX driver can do this in two different ways:

1. Via batch buffer executed in h/w (LRI or LRMs)
2. Direct writes via GFX MMIO

Direct writes using GFX MMIO requires clock gating to be turned off as h/w might already be done with execution. That will require x9424[0] to be cleared prior to register updates and need to be set post register updates.

Note that h/w registers will clear with any reset, hence error logs will have to be re-programmed anything driver is performing a h/w reset or driver is getting re-loaded.

In case of GT3, both physical slices share the same MMIO addressing however their error registers may require different programming values. S/W has to use re-direction register when isolating slice#0 vs slice#1 register spaces. Re-direction register is located at x20CC.

Managing H/W Context via WABB

The error LOG registers are not part of the h/w context and they need to be handled via WBB (workaround batch buffer). Driver has to prepare (and keep up-to-date) a WBB which the h/w would execute every RC6 exit. Driver will add/change the LOG register updates while keeping the WBB up-to-date.

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 be 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).

Description:

- Opcodes: All opcodes except DX11 special opcodes.
- Datatype: All datatypes except bytes.
- Accessmode:
 - Align1:
 - No Scattering or Gathering data. This means data in source and destination registers are aligned and packed (data is contiguous in a register).


```
//Example:
// allowed, data is contiguous and source and destination regioning map one to one.
mov (8) r10.0:f r11.0<8;8,1>:f

// not allowed, data from source is strided and requires gathering to write to destination
mov (8) r10.0:f r11.0<4;4,2>:f

// not allowed, data from source is contiguous but not aligned with destination.
//Destination register requires scattering
mov (8) r10.0<2>:w r11.0<8;8,1>:w

//not allowed, data from source is contiguous but destination is not aligned to source
mov (8) r10.1:f r11.0<4;4,1>:f

// allowed. Source and destination have stride but are aligned
mov (4) r10.1:f r11.1<4;4,1>:f
```
 - A scalar on 32b/64b datatype is allowed.
 - Align16
- Addressmode: Direct Addressing
- Register File: GRF/NULL/Immediates. No access to Accumulator.

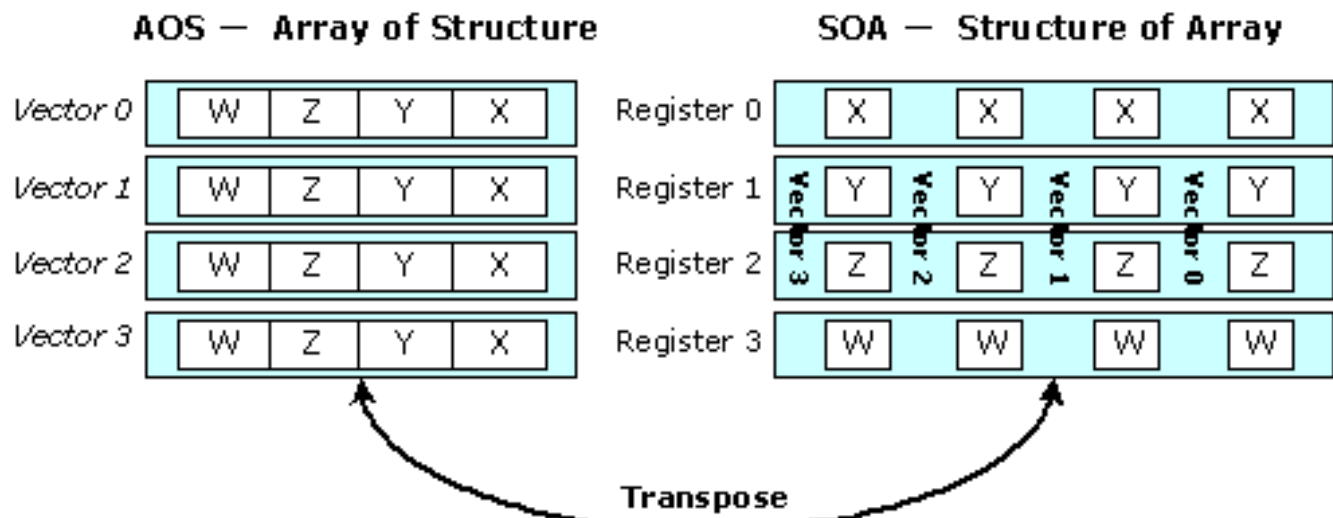
Primary Usage Models

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



B6890-01

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 $\text{SIMD}m \times n$, where m is the size of the vector and n is the number of concurrent program flows executed in SIMD.

Execution with AOS data structures:

- **SIMD4** (short for SIMD4x1) indicates that a SIMD instruction operates on 4-element vectors stored in registers. There is one program flow.
- **SIMD4x2** indicates that a SIMD instruction operates on a pair of 4-element vectors in registers. There are effectively two programs running side by side with one vector per program.

Execution with SOA data structures, also referred to as “channel serial” execution, mostly uses:

- **SIMD8** (short for SIMD1x8) indicates a SIMD instruction based on the SOA data structure where one register contains one data element (the same one) for each of 8 vectors. Effectively, there are 8 concurrent program flows.
- **SIMD16** (short for SIMD1x16) indicates that a SIMD instruction operates on a pair of registers that contain one data element (the same one) for each of 16 vectors. SIMD16 has 16 concurrent program flows.

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}
```

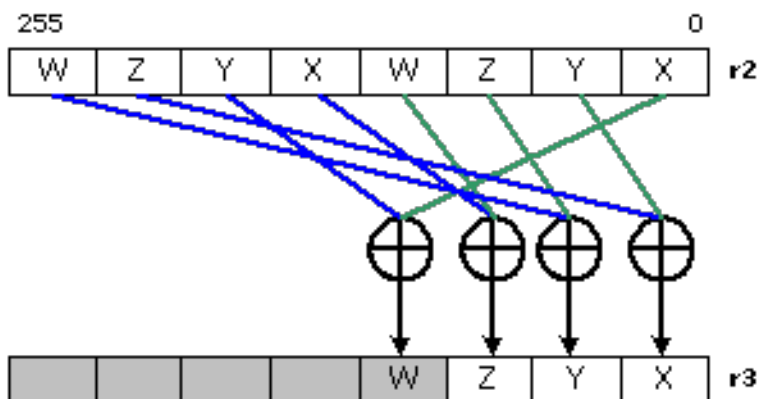
Without diving too much into the syntax definition of a GEN instruction, it is clear that a GEN instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar 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



B 6891-01

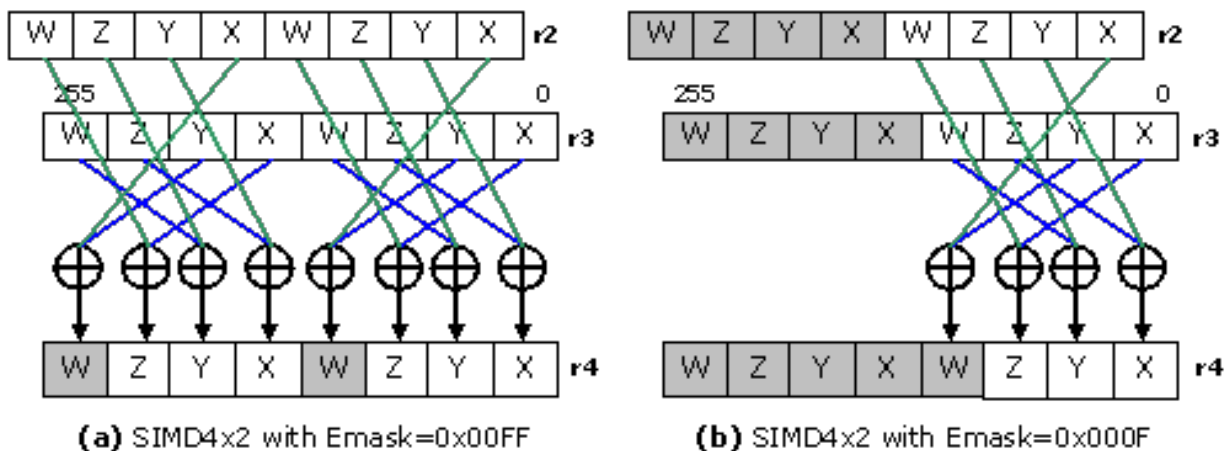
SIMD4x2 Mode of Operation

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

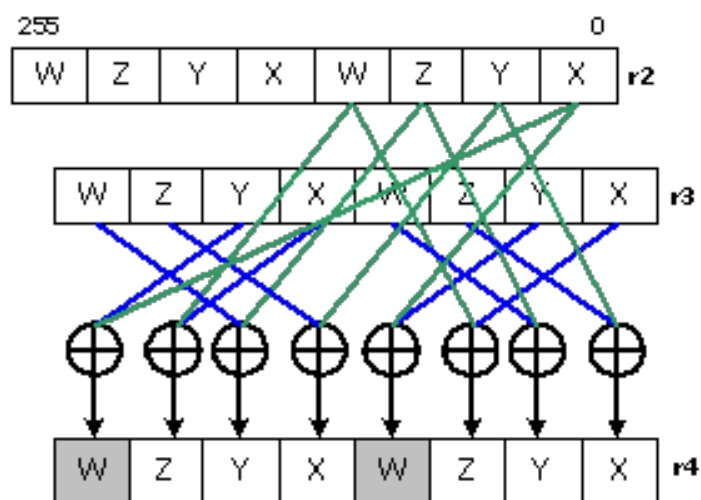


B6892-01

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 Operation*.

A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows

B 6893-01

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
```

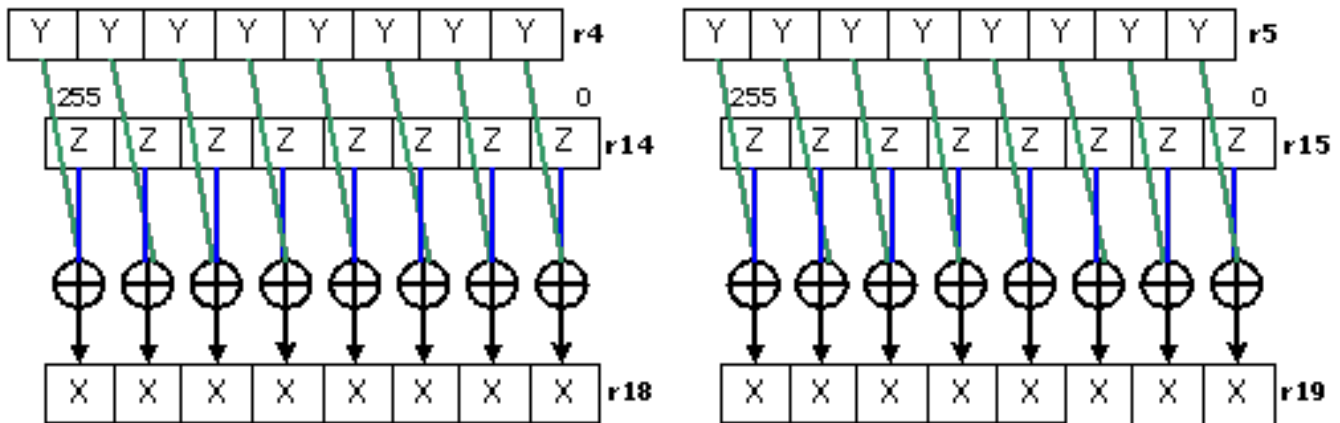
```
add (16) r20<1>:f r6<8;8,1>:f r16<8;8,1>:f // dst.y = src0.z + src1.w
```

```
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



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

B6894-01

Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS format). The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

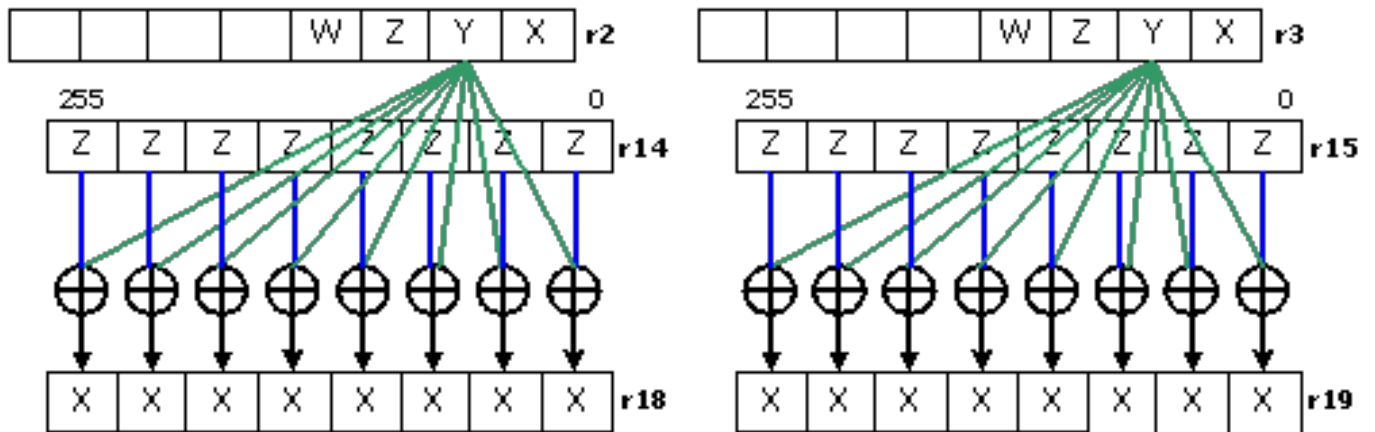
```
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr} // dst.x = src0.y + src1.z
```

```
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



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

B6895-01

SIMD8 Mode of Operation

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:

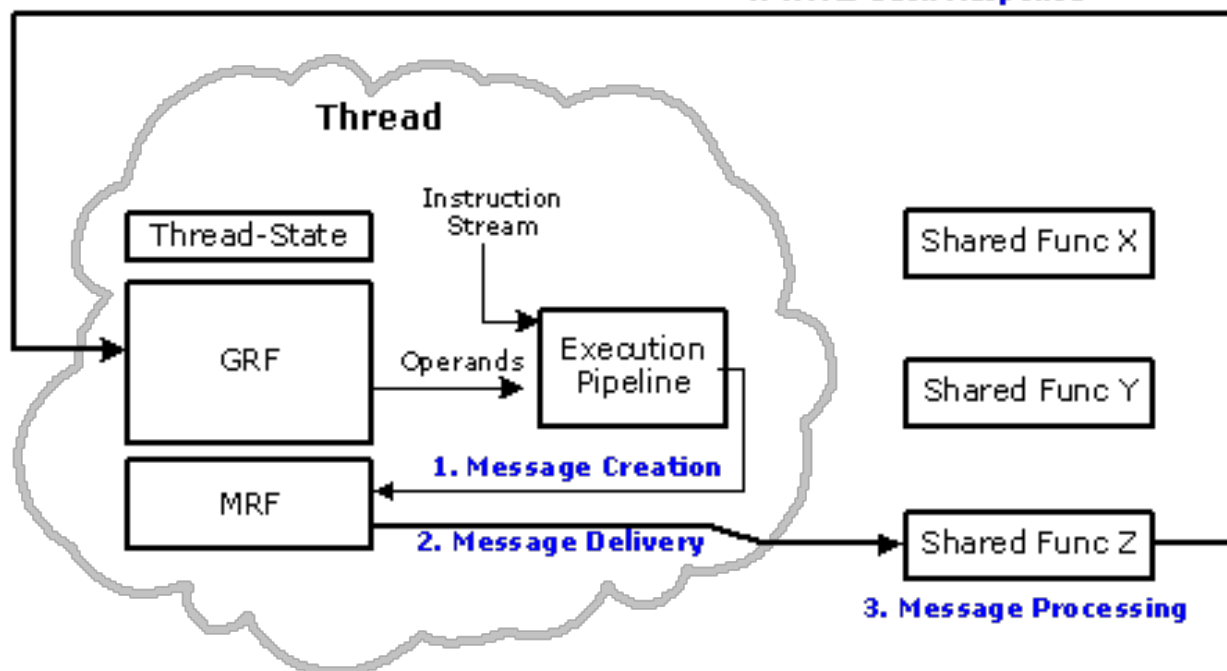
- **Message Payload.**
- Associated (“sideband”) information provided by:
 - **Message Descriptor.** Specified with the send instruction. Included in the message descriptor is control and routing information such as the target function ID, message payload length, response length, etc.
 - Additional information 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

4. Write-back Response



B6876-01

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 <u>not</u> supporting EOT messages
Null, DataPortWrite, URB, MessageGateway, ThreadSpawner	DataPortRead, 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 by snooping all message transmissions, regardless of the explicit destination, looking for messages which signal end-of-thread. The Thread Spawner in the media pipeline does not snoop for EOT. As it is also a shared function, all threads generated by Thread Spawner must send a message to Thread Spawner to explicitly 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 Dataport 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 Read or Write Dataport; or, (c) the Gateway, 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 to the "null" shared function.

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 Incorrect message length	The destination shared function detects unknown opcodes (as specified in the 'send' instructions <desc> field), and known opcodes where the message payload is either too long or too short, and treats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through an interrupt mechanism then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.
Incorrect response length	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 hang 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

This topic is currently under development.

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	<i>null</i>	1	Null register
0001b	<i>a0.#</i>	1	Address register
0010b	<i>acc#</i>	10	Accumulator register
0011b	<i>f#.#</i>	2	Flag register
0100b	<i>ce#</i>	1	Channel Enable register
0101b	<i>msg#</i>	32	Message Control Register
0110b	<i>sp</i>	1	Stack Pointer Register
0111b	<i>sr0.#</i>	1	State register
1000b	<i>cr0.#</i>	1	Control register
1001b	<i>n#</i>	2	Notification Count register
1010b	<i>ip</i>	1	Instruction Pointer register
1011b	<i>tdr</i>	1	Thread Dependency register
1100b	<i>tm0</i>	2	TimeStamp register
1101b	<i>fc#.#</i>	39	Flow Control register
1110b	<i>Reserved</i>		Reserved

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

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

Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF.

The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to generate the destination write mask at byte level.

Subregister fields of an ARF register may not all be populated (indicated by the subregister being indicated as reserved). Writes to unpopulated subregisters are dropped; there are no side effect. Reads from unpopulated subregisters, if not specified, return unpredictable data.

Some ARF registers are read-only. Writes to read-only ARF registers are dropped and there are no side effects.

Null Register

Null Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control?	N/A
Indexable?	No

The null register is a special encoding for an operand that does not have a physical mapping. It is primarily used in instructions to indicate non-existent operands. Writing to the null register has no side effect. Reading from the null register returns an undefined result.

The null register can be used where a source operand is absent. For example, for a single source operand instruction such as MOV or NOT, the second source operand src1 must be a null register.

When the null register is used as the destination operand of an instruction, it indicates the computed results are not stored in any registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, updates to the selected flag register also occur. In this case, the register region fields of the 'null' operand are valid.

Another example use is to use the null register as the posted destination of a *send* instruction for data write to indicate that no write completion acknowledgement is required. In this case, however, the register region fields are still valid. The null register can also be the first source operand for a *send* instruction indicating the absent of the implied move. See the *send* instruction for details.

Address Register

Address Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0001b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	16
Element Size:	16 bits
Element Type:	UW or UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	N/A
Indexable?	No

Description
<p>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 <i>send</i> instruction.</p>

Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000b = a0 All other encodings are reserved.	<p>When register a0 or subregisters in a0 are used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned.</p> <p>00000b = a0.0:uw</p> <p>00010b = a0.1:uw</p> <p>00100b = a0.2:uw</p> <p>00110b = a0.3:uw</p> <p>01000b = a0.4:uw</p> <p>01010b = a0.5:uw</p> <p>01100b = a0.6:uw</p> <p>01110b = a0.7:uw</p> <p>10000b = a0.8:uw</p> <p>10010b = a0.9:uw</p> <p>10100b = a0.10:uw</p> <p>10110b = a0.11:uw</p> <p>11000b = a0.12:uw</p> <p>11010b = a0.13:uw</p> <p>11100b = a0.14:uw</p> <p>11110b = a0.15:uw</p> <p>All other encodings are reserved.</p> <p>However, when register a0 or subregisters in a0 are explicit source and/or destination registers, other data types are allowed as long as the register region falls in the 128-bit range.</p>

Address Register Fields

DWord	Bits	Description
7	31:16	Address subregister a0.15:uw. Follows the same format as a0.3 .
	15:0	Address subregister a0.14:uw. Follows the same format as a0.2 .
6	31:16	Address subregister a0.13:uw. Follows the same format as a0.3 .
	15:0	Address subregister a0.12:uw. Follows the same format as a0.2 .
5	31:16	Address subregister a0.11:uw. Follows the same format as a0.3 .
	15:0	Address subregister a0.10:uw. Follows the same format as a0.2 .
4	31:16	Address subregister a0.9:uw. Follows the same format as a0.3 .
	15:0	Address subregister a0.8:uw. Follows the same format as a0.2 .
3	31:16	Address subregister a0.7:uw. Follows the same format as a0.3 .
	15:0	Address subregister a0.6:uw. Follows the same format as a0.2 .
2	31:16	Address subregister a0.5:uw. Follows the same format as a0.3.
	15:0	Address subregister a0.4:uw. Follows the same format as a0.2.
1	31:16	Address subregister a0.3:uw. This field, 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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> instruction, it provides the higher 16 bits of <desc>. 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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> instruction, it provides the lower 16 bits of <desc>. 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
```

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.

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 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.

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	Project	Security
ARF Register Type Encoding (RegNum[7:4]):	0010b		
Number of Registers:	10	BDW	
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[4:0]
0000b-1001b = acc0-acc9 All other encodings are reserved.	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.

-

Limits on SIMD16 Float Operations
Accumulator registers may be accessed explicitly as src0 operands only.

- Swizzling is not allowed when an accumulator is used as an implicit source or an explicit source in an instruction.
- Reading accumulator content with a datatype different from the previous write will result in undeterministic values.
- Word datatype write to accumulator is not allowed when destination is odd offset strided by 2.
- For any DWord operation, including DWord multiply, accumulator can store up to 8 channels of data, with only acc0 supported.
- When an accumulator register is an explicit destination, it follows the rules of a destination register. If an accumulator is an explicit source operand, its register region must match that of the destination register with the exception(s) described below.

Exceptions
When HalfFloats access accumulator, interleaved reads/writes is not allowed. This means, accumulator must be accessed in pairs of half-floats.
When OWords of accumulators are accessed, the source and destination OWords may be different. For example, the following instructions are allowed: <pre>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</pre> 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.
Accumulator should not be used as destination for double-float to float format conversion.
When HalfFloats access accumulator, use of Q3/Q4 quarter control is not allowed.

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 1FFFFFFFFh. 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	N/A	N/A	N/A	Not supported data type.
HF	acc0/acc1	16	16	When accumulator is used for Half Float, it has the exact same precision as any GRF register.
Q	N/A	N/A	N/A	Not supported data type.
D (UD)	acc0	8	33/64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword integer values. The data are always stored in accumulator in 2's complement form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting of <i>mul</i> , <i>mach</i> , and <i>shr</i> (or <i>mov</i>).
W (UW)	acc0	16	33	When the internal execution data type is word integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's complement form with 33 bits total. This supports single instruction multiplication of two word sources in W and/or UW format.
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.
- 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.

Channel Select/Channel Enable Encoding for Implicit Source/Destination Access

Encoding	Accumulator Register
00000000b	acc2
00000001b	acc3
00000010b	acc4
00000011b	acc5
00000100b	acc6
00000101b	acc7
00000110b	acc8
00000111b	acc9
00001000b	noacc

On a context save, the sequence of operations shown below MUST be used to save acc2-acc9 registers.

```

mov(8) r113:ud acc2:ud    { NoMask } //acc2
mov(8) r114:ud acc2.yx:ud {Align16}  //acc3
mov(8) r115:ud acc2.zx:ud {Align16}  //acc4
mov(8) r116:ud acc2.wx:ud {Align16}  //acc5

mov(8) r113:ud acc2.xy:ud {Align16}  //acc6
mov(8) r114:ud acc2.yy:ud {Align16}  //acc7
mov(8) r115:ud acc2.zy:ud {Align16}  //acc8
mov(8) r116:ud acc2.wy:ud {Align16}  //acc9

```

On a context restore, the sequence of operations shown below MUST be used to restore acc2-acc9 registers.

```

mov(8) acc2:ud r100:ud    { NoMask } //acc2
mov(8) acc2.x:ud r101:ud  {Align16}  //acc3
mov(8) acc2.y:ud r102:ud  {Align16}  //acc4
mov(8) acc2.xy:ud r103:ud {Align16}  //acc5

mov(8) acc2.z:ud r100:ud  {Align16}  //acc6
mov(8) acc2.xz:ud r101:ud {Align16}  //acc7
mov(8) acc2.yz:ud r102:ud {Align16}  //acc8
mov(8) acc2:ud r103:ud    {Align16}  //acc9

```

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 halves, quarters, or eighths using the QtrCtrl and NibCtrl instruction fields. Those fields affect the selection of flags for predication and conditional modifiers, but do not affect reading or writing flags as explicit instruction operands.

The values held in the individual bits of a flag register are the result of the most recent instruction with a conditional modifier and specifying that flag register. For example:

```
add.nz.f0.0 ...
```

Updates flag subregister f0.0 with the per-channel results of the not zero condition.

The flag register has per-bit write enables. When being updated as the secondary destination associated with a conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. Other bits in the flag subregister are unchanged.

Flag registers and subregisters can also be explicit source or destination operands.

The *sel* instruction does not update flags.

Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000b = f0:ud	00000b = fn.0:uw
0001b = f1:ud	00010b = fn.1:uw
Other encodings are reserved.	Other encodings are reserved.

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

Channel Enable Register Fields

DWord	Bits	Description
0	31:0	Channel Enable Register ce0.0: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 mode 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.

On a context save, the message control register is written to the upper oword of a register.

On a context restore, the upper oword of register must be moved to message control register. However, since these move operations require data from lower oword of a register, it is required to move the upper oword to lower oword before moving to these registers.

Example

Context Save Cycle:

```
mov (8) r113.0:ud msg0.0:ud { NoMask }
```

The above operation saves content of fc0.0 to the upper oword of r113

Context Restore Cycle:

```
mov (4) rtmp.0:ud r113.4<4;4,1>:ud { NoMask }
mov (8) msg0.0:ud rtmp:ud { NoMask }
```

The above operation restores content from upper oword of r113 by moving to lower oword of rtmp.

Programming Note	
Context:	Message Control Registers
Message Control Registers must never be saved or restored	

SP Register

SP Register Summary

Attribute	Value	Project
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 SP register can be accessed as a unsigned DWord integer. It is a read-write register, containing the current stack pointer, which is relative to the Generate State Base Address. 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.

Register and Subregister Numbers for SP Register

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

SP Register Fields

DWord	Bits	Description
0..1	63:48	Reserved. MBZ.
	47:0	<p>sp. Specifies the current stack pointer. This pointer is relative to the General State Base Address. This register is initialized at thread load to the top of the per thread Scratch Space. The register is R/W.</p> $sp = [scratch\ space\ pointer] + [scratch\ space] - 1$ <p>Alternatively, this register may be updated by Software to any flat address space. In such cases, the stack is NOT relative to the General State Base Address. Software must ensure that the address is exclusive for the thread.</p>
2..3	63:48	Reserved. MBZ.
	47:0	<p>sp limit. Specifies the upper limit for the stack pointer. This pointer is relative to the General State Base Address. This register is initialized at thread load to the limit allocated for stack in the state. See the GPGPU Thread Payload description for details. The register is R/W.</p> $sp_limit = [scratch\ space\ pointer] + [stack\ space\ limit]$ <p>Alternatively, this register may be updated by software, similar to the sp register. In such cases, software is responsible for allocating the right thread stack pointer limit.</p>

State Register

State Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0111b
Number of Registers:	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

On a context restore, SIP must ensure all pending context restore cycles to the GRF are committed/completed before restoring SR1.

Example:

```

  send (8)  rdest0 rsrc0 <msg_dscpt>
send (8)  rdest1 rsrc1 <msg_dscpt>
send (8)  rdest2 rsrc2 <msg_dscpt>
send (8)  rdest3 rsrc3 <msg_dscpt>
send (8)  rdest4 rsrc4 <msg_dscpt>
// the following mov's are required to ensure all previous reads are complete
mov (8) null rdest0
mov (8) null rdest1
mov (8) null rdest2
mov (8) null rdest3
mov (8) null rdest4
// restore the grfsb
mov (8) srl rsrc

```

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	<p>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.</p> <p>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.</p> <p>0 = Low priority class. 1 = High priority class.</p>
	22:19	Reserved. MBZ.
	18:16	<p>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.</p> <p>This field is set to zero at a thread's dispatch.</p> <p>During a thread's run time, this field may or may not be incremented when a new thread is dispatched to the same EU. It is only incremented when another thread's priority number is incremented and reaches the same value. For example, if currently there is a thread with priority 0 on an EU, then dispatching a new thread to that EU causes the old thread's priority number to increment to 1. However, if the active thread (assuming for simplicity that there is only one) on an EU has a priority number 1 (or 2 or 3), then dispatching a new thread to this EU does not change the old thread's priority number. As threads on an EU may terminate in arbitrary order, the exact number for a thread depends on the dynamic execution of threads.</p> <p>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.</p>
	15:8	<p>[15:14] Slice ID. [13:12] SubSlice ID. [11:8] EUID.</p>
	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.
1 (sr0.1:ud)	31:23	FFTID (Fixed Function Thread ID). There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.
	22	Reserved.
	21	Hardware Defined State. The byte is defined for hardware use only. The content is saved and restored in the event of mid thread preemption.

DWord	Bits	Description													
	20	Page Fault Status. This bit specifies if the thread has hit a page fault on a memory read, memory write or instruction fetch operation. The bit is cleared when the thread is restarted after a fault.													
	19	Reserved.													
	18:16	Page Fault Code. The fault code indicates the type of fault encountered.													
	15:8	Hardware Defined State. The byte is defined for hardware use only. The content is saved and restored in the event of mid thread preemption.													
	7:0	IEEE Exception. The exception bits are sticky bits set by the opcode when one of the exception is triggered. These bits are defined per thread and all channels update one sticky bit. These bits may be cleared by software or on a thread load. Updates to these bits may be turned OFF by the IEEE Exception trap enable in the CR register. When these bits are required as source of an operation, the previous instruction must use a {Switch}. This ensures all asynchronous flag updates are complete before using as source operand. The following table describes these bits: <table><tr><th>Bits</th><th>Definition</th></tr><tr><td>[7:5]</td><td>Reserved</td></tr><tr><td>4</td><td>Inexact Exception</td></tr><tr><td>3</td><td>Overflow</td></tr><tr><td>2</td><td>Underflow</td></tr><tr><td>1</td><td>Divide by Zero</td></tr><tr><td>0</td><td>Invalid Operation</td></tr></table>	Bits	Definition	[7:5]	Reserved	4	Inexact Exception	3	Overflow	2	Underflow	1	Divide by Zero	0
Bits	Definition														
[7:5]	Reserved														
4	Inexact Exception														
3	Overflow														
2	Underflow														
1	Divide by Zero														
0	Invalid Operation														
	20	Page Fault Status. This bit specifies if the thread has hit a page fault on a memory read, memory write or instruction fetch operation. The bit is cleared when the thread is restarted after a fault.													
	19	Reserved.													
	18:16	Page Fault Code. The fault code indicates the type of fault encountered.													
	15:7	Hardware Defined State. The byte is defined for hardware use only. The content is saved and restored in the event of mid thread preemption.													
	6:0	IEEE Exception. The exception bits are sticky bits set by the opcode when one of the exception is triggered. These bits are defined per thread and all channels update one sticky bit. These bits may be cleared by software or on a thread load. Updates to these bits may be turned OFF by the IEEE Exception trap enable in the CR register. When these bits are required as source of an operation, the previous instruction must use a {Switch}. This ensures all asynchronous flag updates are complete before using as source operand. The following table describes these bits: <table><tr><th>Bits</th><th>Definition</th></tr><tr><td>[6:5]</td><td>Reserved</td></tr><tr><td>4</td><td>Inexact Exception</td></tr><tr><td>3</td><td>Overflow</td></tr><tr><td>2</td><td>Underflow</td></tr><tr><td>1</td><td>Divide by Zero</td></tr></table>	Bits	Definition	[6:5]	Reserved	4	Inexact Exception	3	Overflow	2	Underflow	1	Divide by Zero	
		Bits	Definition												
		[6:5]	Reserved												
		4	Inexact Exception												
		3	Overflow												
		2	Underflow												
		1	Divide by Zero												

DWord	Bits	Description														
		<table><tr><td>0</td><td>Invalid Operation</td></tr></table>	0	Invalid Operation												
0	Invalid Operation															
	21	Reserved.														
	20	Page Fault Status. This bit specifies if the thread has hit a page fault on a memory read, memory write or instruction fetch operation. The bit is cleared when the thread is restarted after a fault.														
	18:16	Page Fault Code. The fault code indicates the type of fault encountered.														
	15:7	Hardware Defined State. The byte is defined for hardware use only. The content is saved and restored in the event of mid thread preemption.														
	6:0	IEEE Exception. The exception bits are sticky bits set by the opcode when one of the exception is triggered. These bits are defined per thread and all channels update one sticky bit. These bits may be cleared by software or on a thread load. Updates to these bits may be turned OFF by the IEEE Exception trap enable in the CR register. When these bits are required as source of an operation, the previous instruction must use a {Switch}. This ensures all asynchronous flag updates are complete before using as source operand. The following table describes these bits:														
		<table><tr><th>Bits</th><th>Definition</th></tr><tr><td>[6:5]</td><td>Reserved</td></tr><tr><td>4</td><td>Inexact Exception</td></tr><tr><td>3</td><td>Overflow</td></tr><tr><td>2</td><td>Underflow</td></tr><tr><td>1</td><td>Divide by Zero</td></tr><tr><td>0</td><td>Invalid Operation</td></tr></table>	Bits	Definition	[6:5]	Reserved	4	Inexact Exception	3	Overflow	2	Underflow	1	Divide by Zero	0	Invalid Operation
		Bits	Definition													
		[6:5]	Reserved													
		4	Inexact Exception													
		3	Overflow													
		2	Underflow													
		1	Divide by Zero													
0	Invalid Operation															
2 (sr0.2:ud)	31:0	Dispatch Mask (DMask). This 32-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register. Format: U32														
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. 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.

The SIP is widened to 48 bits. However the EU still only uses the low 32 bits.

Register and Subregister Numbers for Control Register

RegNum[3:0]	SubRegNum[4:0]
0000b = <i>cr0</i> 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	Reserved. MBZ.
	11	Reserved. MBZ.
	10	Reserved. MBZ.
	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 <i>not</i> 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)
	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)

DWord	Bits	Description
	2	<p>Single Program Flow (SPF). Specifies whether the thread has a single program flow (SIMDn_{xm} with m = 1) or multiple program flows (SIMDn_{xm} with m > 1). This bit affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This bit is initialized by the Thread Dispatch.</p> <p>0: Multiple Program Flows 1: Single Program Flow</p> <p>Programming Restrictions: Only H1/Q1/N1 are allowed in SPF mode. Power Optimization: If an entire shader does not do SIMD branching, the driver can set the SPF bit to 1 to save power in HW. SPF mode must be set to 0 at the start of context restore.</p>
	1	<p>Accumulator Disable. This bit controls the update of the accumulator by the instruction field AccWrCtrl. If this bit is cleared, the accumulator is updated for all instructions with AccWrCtrl enabled. If set, the accumulator is disabled for all update operations, maintaining its value prior to being disabled. Setting this bit has no effect if the accumulator is the explicit destination operand for an instruction. This bit is initialized to 0.</p> <p>0: Enable accumulator update. 1: Disable accumulator update.</p> <p>Usage Notes: 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.</p>
	0	<p>Single Precision Floating Point Mode (FP Mode). This bit specifies whether the current single-precision floating-point operation mode is IEEE mode (IEEE Standard 754) or the ALT (alternative mode). This bit does not affect the floating-point mode used for other floating-point data types. This bit is also forwarded on the message sideband for all out-going messages, for example, to control the floating-point mode of the Sampler. Software may modify this bit to dynamically switch between the two floating-point modes. This bit is initialized by Thread Dispatch.</p> <p>0 = IEEE floating-point mode for the F (Float) type. 1 = ALT (alternative) floating-point mode for the F (Float) type.</p>
	30	<p>External Halt Exception Status and Control. This bit indicates the External Halt exception. It is set by EU hardware on receiving the broadcast External Halt signal. The System Routine should reset this bit before returning to an application routine to avoid infinite loops. This bit may be set or cleared by software. This bit is initialized to 0.</p>

DWord	Bits	Description
	29	Software Exception Control. This bit is the control bit for software exceptions. Setting this bit to 1 in an application routine causes an exception. Clearing this bit in an application routine has no effect. Upon entering the system routine, the hardware maintains this bit as 1 to signify a software exception. The System Routine should reset this bit before returning to an application routine. This bit may be set or cleared by software. This bit is initialized to 0.
	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.
	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

DWord	Bits	Description								
	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. <table><tr><th>Exception Type</th><th>AIP Value</th></tr><tr><td>External Halt</td><td>N/A ⁽¹⁾</td></tr><tr><td>Software Exception</td><td>IP + 1</td></tr><tr><td>Illegal Opcode</td><td>IP</td></tr></table> <p>(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.</p>	Exception Type	AIP Value	External Halt	N/A ⁽¹⁾	Software Exception	IP + 1	Illegal Opcode	IP
Exception Type	AIP Value									
External Halt	N/A ⁽¹⁾									
Software Exception	IP + 1									
Illegal Opcode	IP									
	2:0	Reserved. MBZ.								

Programming Note

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.

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

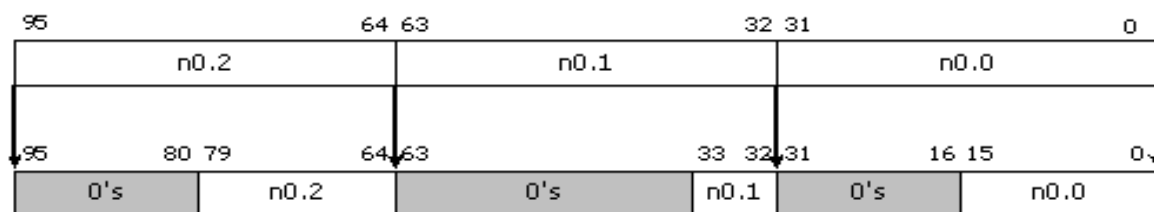
Notification Register 1 Fields

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 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



B6898-01

IP Register

IP Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

The ip register can be accessed as a 32-bit quantity. It is a read-write register, containing the current instruction pointer, which is relative to the **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. When it is written, the 3 LSBs are dropped by hardware.

Register and Subregister Numbers for IP Register

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

IP Register Fields

DWord	Bits	Subfield Description
0	31:3	Ip. Specifies the current instruction pointer. This pointer is relative to the 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:	8
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.

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
	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:	3
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 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).	

Register and Subregister Numbers for Performance Register

RegNum[3:0]	SubRegNum[4:0]
0000b = <i>tm0</i> 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	Undefined Format: U32

Flow Control Registers

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 Control Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = fc0	00000b-11111b = fc0.0–fc0.31 .
0001b = fc1	00000b = fc1.0 . All other encodings are reserved.
0010b = fc2	00000b = fc2.0 . All other encodings are reserved.
0011b = fc3	00000b = fc3.0 . 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.

When restoring to fc4.0, the DW0 of the register must be copied to DW1 of the register before restoring to fc4.0.

Example:

```
mov (8) fc4.0 r4.0
```

This must be replaced with:

```
mov (1) r4.1:ud r4.0:ud
mov (8) fc4.0 r4.0
```

Once context save of fc0 is done, 0 must be copied to fc0 register.

On a context save, the flow control register is written to the upper oword of a register.

On a context restore, the upper oword of register must be moved to flow control register. However, since these move operations require data from lower oword of a register, it is required to move the upper oword to lower oword before moving to these registers.

Example:

Context Save Cycle:

```
mov      (8)  r113.0:ud      fc0.0:ud      { NoMask }
```

The above operation saves content of fc0.0 to the upper oword of r113.

Context Restore Cycle:

```
mov      (4)  rtmp.0:ud r113.4<4:4,1>:ud      { NoMask }
mov      (8)  fc0.0:ud rtmp:ud                  { NoMask }
```

The above operation restores content from upper oword of r113 by moving to lower oword of rtmp.

On a context restore, the sequence of operations shown below MUST be used to restore Flow Control Register. The registers used are as an example.

```
// Restore fc0.28 to fc0.31 into registers r100 to r103 (Send operation)
mov      (4)  r100.0:ud r100.4<4:4,1>:ud      { NoMask }
mov      (4)  r101.0:ud r101.4<4:4,1>:ud      { NoMask }
mov      (4)  r102.0:ud r102.4<4:4,1>:ud      { NoMask }
mov      (4)  r103.0:ud r103.4<4:4,1>:ud      { NoMask }

mov      (2)  r104.1:ud r100.0<2:2,1>:ud { NoMask }
mov      (2)  r105.1:ud r101.0<2:2,1>:ud { NoMask }
mov      (2)  r106.1:ud r102.0<2:2,1>:ud { NoMask }
mov      (2)  r107.1:ud r103.0<2:2,1>:ud { NoMask }

mov      (8)  fc0.28:ud r104:ud      { NoMask }
mov      (8)  fc0.29:ud r105:ud      { NoMask }
mov      (8)  fc0.30:ud r106:ud      { NoMask }
mov      (8)  fc0.31:ud r107:ud      { NoMask }

// Restore fc0.24 to fc0.27 into registers r100 to r103 (Send operation)
mov      (4)  r100.0:ud r100.4<4:4,1>:ud      { NoMask }
mov      (4)  r101.0:ud r101.4<4:4,1>:ud      { NoMask }
mov      (4)  r102.0:ud r102.4<4:4,1>:ud      { NoMask }
mov      (4)  r103.0:ud r103.4<4:4,1>:ud      { NoMask }

mov      (2)  r104.2:ud r100.0<2:2,1>:ud { NoMask }
mov      (2)  r105.2:ud r101.0<2:2,1>:ud { NoMask }
mov      (2)  r106.2:ud r102.0<2:2,1>:ud { NoMask }
mov      (2)  r107.2:ud r103.0<2:2,1>:ud { NoMask }

mov      (8)  fc0.24:ud r104:ud      { NoMask }
mov      (8)  fc0.25:ud r105:ud      { NoMask }
mov      (8)  fc0.26:ud r106:ud      { NoMask }
mov      (8)  fc0.27:ud r107:ud      { NoMask }

// Restore fc0.20 to fc0.23 into registers r100 to r103 (Send operation)
mov      (4)  r100.0:ud r100.4<4:4,1>:ud      { NoMask }
mov      (4)  r101.0:ud r101.4<4:4,1>:ud      { NoMask }
mov      (4)  r102.0:ud r102.4<4:4,1>:ud      { NoMask }
mov      (4)  r103.0:ud r103.4<4:4,1>:ud      { NoMask }
```

```

mov     (2)  r104.3:ud r100.0<2;2,1>:ud { NoMask }
mov     (2)  r105.3:ud r101.0<2;2,1>:ud { NoMask }
mov     (2)  r106.3:ud r102.0<2;2,1>:ud { NoMask }
mov     (2)  r107.3:ud r103.0<2;2,1>:ud { NoMask }

mov     (8)  fc0.20:ud r104:ud           { NoMask }
mov     (8)  fc0.21:ud r105:ud           { NoMask }
mov     (8)  fc0.22:ud r106:ud           { NoMask }
mov     (8)  fc0.23:ud r107:ud           { NoMask }

// Restore fc0.16 to fc0.19 into registers r100 to r103 (Send operation)
mov     (4)  r100.0:ud r100.4<4;4,1>:ud { NoMask }
mov     (4)  r101.0:ud r101.4<4;4,1>:ud { NoMask }
mov     (4)  r102.0:ud r102.4<4;4,1>:ud { NoMask }
mov     (4)  r103.0:ud r103.4<4;4,1>:ud { NoMask }

mov     (2)  r104.4:ud r100.0<2;2,1>:ud { NoMask }
mov     (2)  r105.4:ud r101.0<2;2,1>:ud { NoMask }
mov     (2)  r106.4:ud r102.0<2;2,1>:ud { NoMask }
mov     (2)  r107.4:ud r103.0<2;2,1>:ud { NoMask }

mov     (8)  fc0.16:ud r104:ud           { NoMask }
mov     (8)  fc0.17:ud r105:ud           { NoMask }
mov     (8)  fc0.18:ud r106:ud           { NoMask }
mov     (8)  fc0.19:ud r107:ud           { NoMask }

// Restore fc0.12 to fc0.15 into registers r100 to r103 (Send operation)
mov     (4)  r100.0:ud r100.4<4;4,1>:ud { NoMask }
mov     (4)  r101.0:ud r101.4<4;4,1>:ud { NoMask }
mov     (4)  r102.0:ud r102.4<4;4,1>:ud { NoMask }
mov     (4)  r103.0:ud r103.4<4;4,1>:ud { NoMask }

mov     (2)  r104.5:ud r100.0<2;2,1>:ud { NoMask }
mov     (2)  r105.5:ud r101.0<2;2,1>:ud { NoMask }
mov     (2)  r106.5:ud r102.0<2;2,1>:ud { NoMask }
mov     (2)  r107.5:ud r103.0<2;2,1>:ud { NoMask }

mov     (8)  fc0.12:ud r104:ud           { NoMask }
mov     (8)  fc0.13:ud r105:ud           { NoMask }
mov     (8)  fc0.14:ud r106:ud           { NoMask }
mov     (8)  fc0.15:ud r107:ud           { NoMask }

// Restore fc0.8 to fc0.11 into registers r100 to r103 (Send operation)
mov     (4)  r100.0:ud r100.4<4;4,1>:ud { NoMask }
mov     (4)  r101.0:ud r101.4<4;4,1>:ud { NoMask }
mov     (4)  r102.0:ud r102.4<4;4,1>:ud { NoMask }
mov     (4)  r103.0:ud r103.4<4;4,1>:ud { NoMask }

mov     (2)  r104.6:ud r100.0<2;2,1>:ud { NoMask }
mov     (2)  r105.6:ud r101.0<2;2,1>:ud { NoMask }
mov     (2)  r106.6:ud r102.0<2;2,1>:ud { NoMask }
mov     (2)  r107.6:ud r103.0<2;2,1>:ud { NoMask }

mov     (8)  fc0.8:ud r104:ud            { NoMask }
mov     (8)  fc0.9:ud r105:ud            { NoMask }
mov     (8)  fc0.10:ud r106:ud           { NoMask }
mov     (8)  fc0.11:ud r107:ud           { NoMask }

// Restore fc0.4 to fc0.7 into registers r100 to r103 (Send operation)
mov     (4)  r100.0:ud r100.4<4;4,1>:ud { NoMask }
mov     (4)  r101.0:ud r101.4<4;4,1>:ud { NoMask }
mov     (4)  r102.0:ud r102.4<4;4,1>:ud { NoMask }

```

```

mov    (4)  r103.0:ud r103.4<4;4,1>:ud    { NoMask }

mov    (1)  r104.7:ud r100.0:ud { NoMask }
mov    (1)  r104.0:ud r100.1:ud { NoMask }
mov    (1)  r105.7:ud r101.0:ud { NoMask }
mov    (1)  r105.0:ud r101.1:ud { NoMask }

mov    (1)  r106.7:ud r102.0:ud { NoMask }
mov    (1)  r106.0:ud r102.1:ud { NoMask }
mov    (1)  r107.7:ud r103.0:ud { NoMask }
mov    (1)  r107.0:ud r103.1:ud { NoMask }

mov    (8)  fc0.4:ud r104:ud      { NoMask }
mov    (8)  fc0.5:ud r105:ud      { NoMask }
mov    (8)  fc0.6:ud r106:ud      { NoMask }
mov    (8)  fc0.7:ud r107:ud      { NoMask }

// Restore fc0.0 to fc0.3 into registers r100 to r103 (Send operation)
mov    (4)  r100.0:ud r100.4<4;4,1>:ud    { NoMask }
mov    (4)  r101.0:ud r101.4<4;4,1>:ud    { NoMask }
mov    (4)  r102.0:ud r102.4<4;4,1>:ud    { NoMask }
mov    (4)  r103.0:ud r103.4<4;4,1>:ud    { NoMask }

mov    (8)  fc0.0:ud r100:ud      { NoMask }
mov    (8)  fc0.1:ud r101:ud      { NoMask }
mov    (8)  fc0.2:ud r102:ud      { NoMask }
mov    (8)  fc0.3:ud r103:ud      { NoMask }

```

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 <i>imm32:type</i> for 32-bit immediates and <i>imm64:type</i> for 64-bit immediates.

Description
For a scalar immediate, the numeric data types supported are <i>:uw, :w, :ud, :d, :uq, :q</i> for integers AND <i>:hf, :f, :df</i> 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.

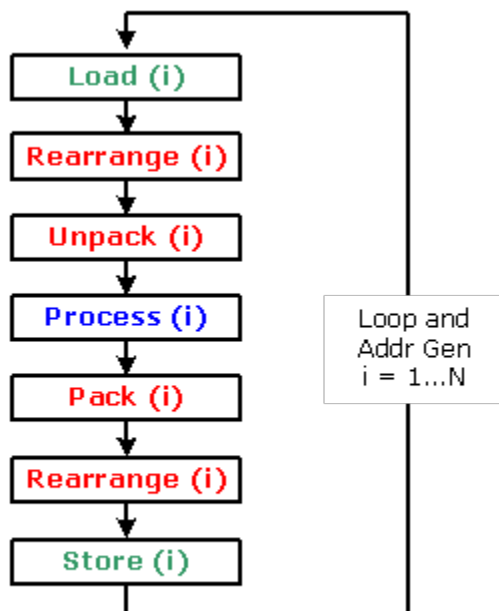
When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (***v***) and equivalent to a DWord for an immediate float vector (***vf***).

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.

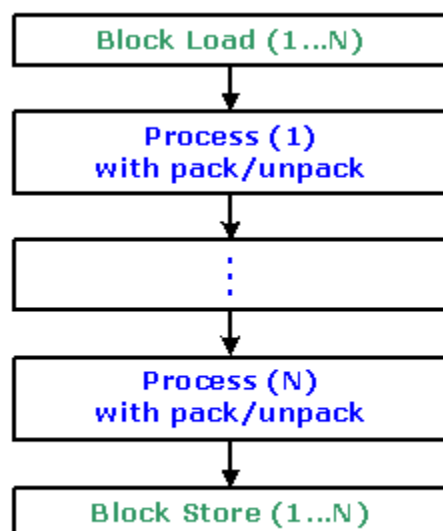
This can be illustrated by the example shown in *Region Parameters* and *Region Parameters*. As shown in *Region Parameters*, a sequence of SIMD instruction is executed on a conventional load/store based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first 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



B.6900-01

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.

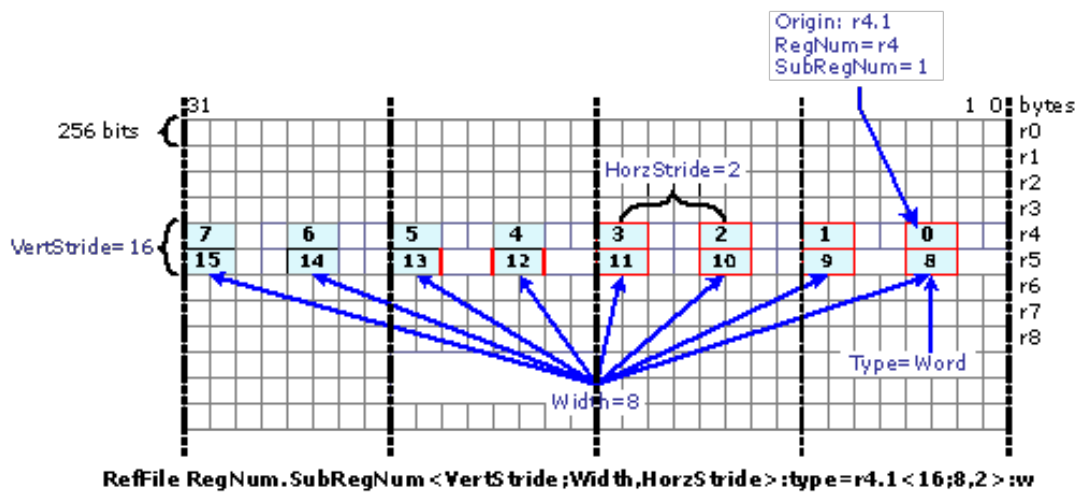
- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
 - Exception of this rule is present for the integer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

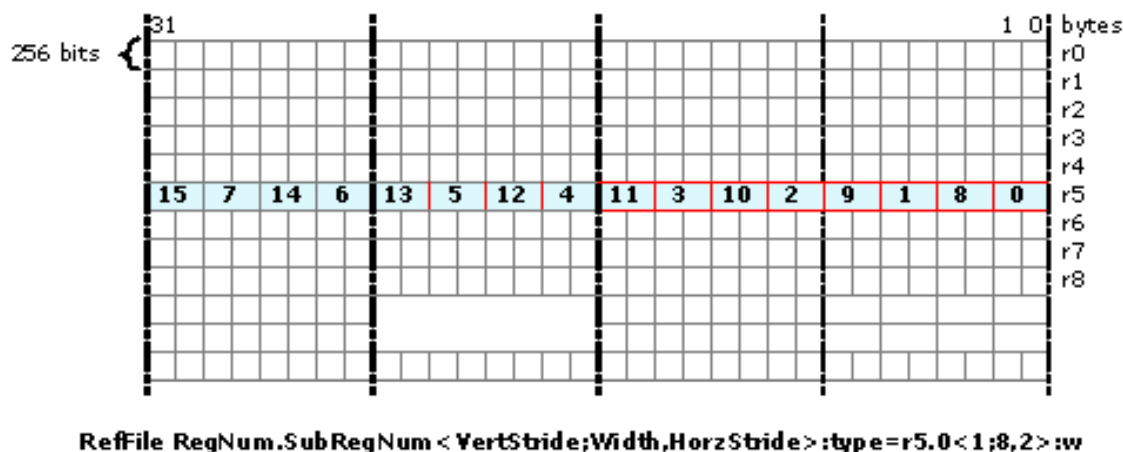


B6901-01

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



B6902-01

Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The

following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.

```
// Input: Type: ub | b | uw | w | ud | d | f | v
//RegNum: In unit of 256-bit register
//SubRegNum: In unit of data element size
//ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels
int ElementSize = (Type=="b"||Type=="ub") ? 1 : (Type=="w"||Type=="uw") ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum<<5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
    int Offset = RowBase;
    for (int x=0; x<Width; x++) {
        Address [Channel++] = Offset;
        Offset += HorzStride*ElementSize;
    }
    RowBase += VertStride * ElementSize;
}
```

As *HorzStride* and *VertStride* are specified independently (note that *VertStride* might be smaller than or equal to *HorzStride*), the region may take various shapes from a replicated scalar, a replicated vector, a vector of replicated scalars, a sliding window, to a non-overlapped 2D array.

A region-based description of a destination operand can take the following simplified format

RegFile RegNum.SubRegNum<HorzStride>.type

The destination operand is only allowed to have a 1 dimensional region. The Register Region Origin and Type are the same as for a source operand. The total number of elements is given by *ExecSize*. However, only *HorzStride* is required to describe the 1D region, not *VertStride* and *Width*.

As a source register region may cross multiple physical GRF registers, an instruction with such source operands may take more than two execution cycles to gather source data elements for execution. The destination register region is restricted to be within a physical GRF register. In other words, destination scatter writes over multiple registers are not supported.

Region Addressing Modes

There are two different register addressing modes: Direct register addressing and register-indirect register addressing. Depending on the register region description, the register-indirect register addressing mode can be further divided into three usages: 1x1 index region where only the origin of register region is provided by the address register, Vx1 index region where the offset of each row of the register region is provided by an address register, VxH index region where the offset of each data element is provided by an address register.

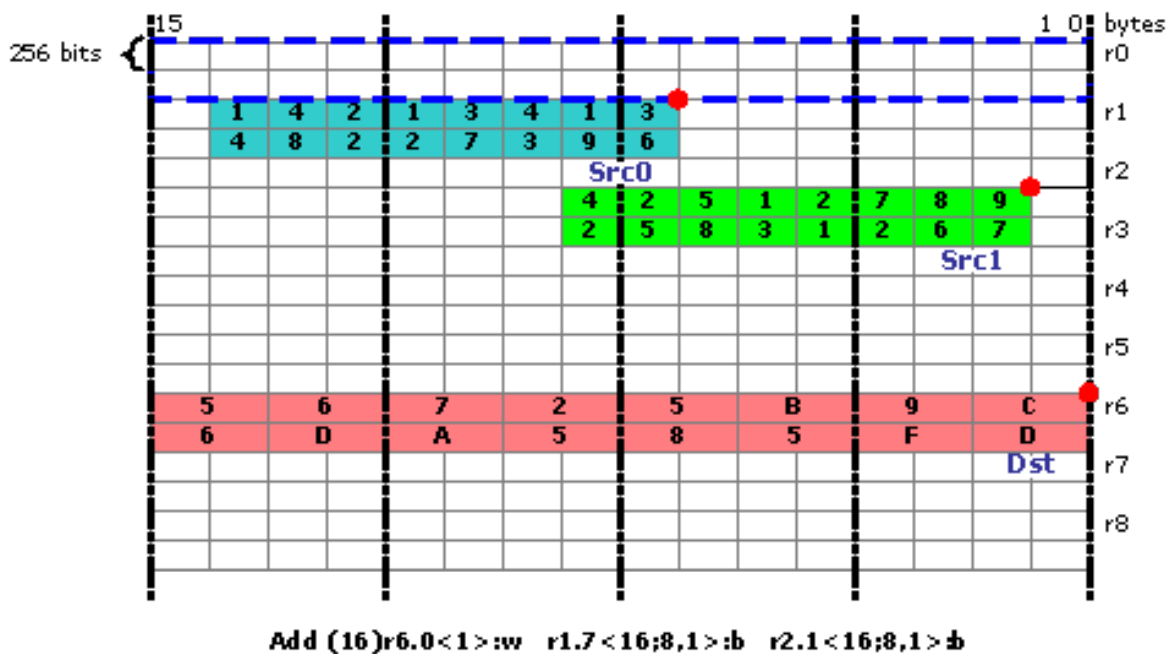
Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Direct Register Addressing and *Direct Register Addressing* are two examples of direct register addressing.

For the example in *Direct Register Addressing*, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical stride of 16 bytes for *Src0* and *Src1* indicates that the first element and the 9'th element are 16 bytes apart in the register file. Note that *Src0* falls into the 256-bit physical GRF register starting at r1.0, but *Src1* crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have $C = 3 + 9$.

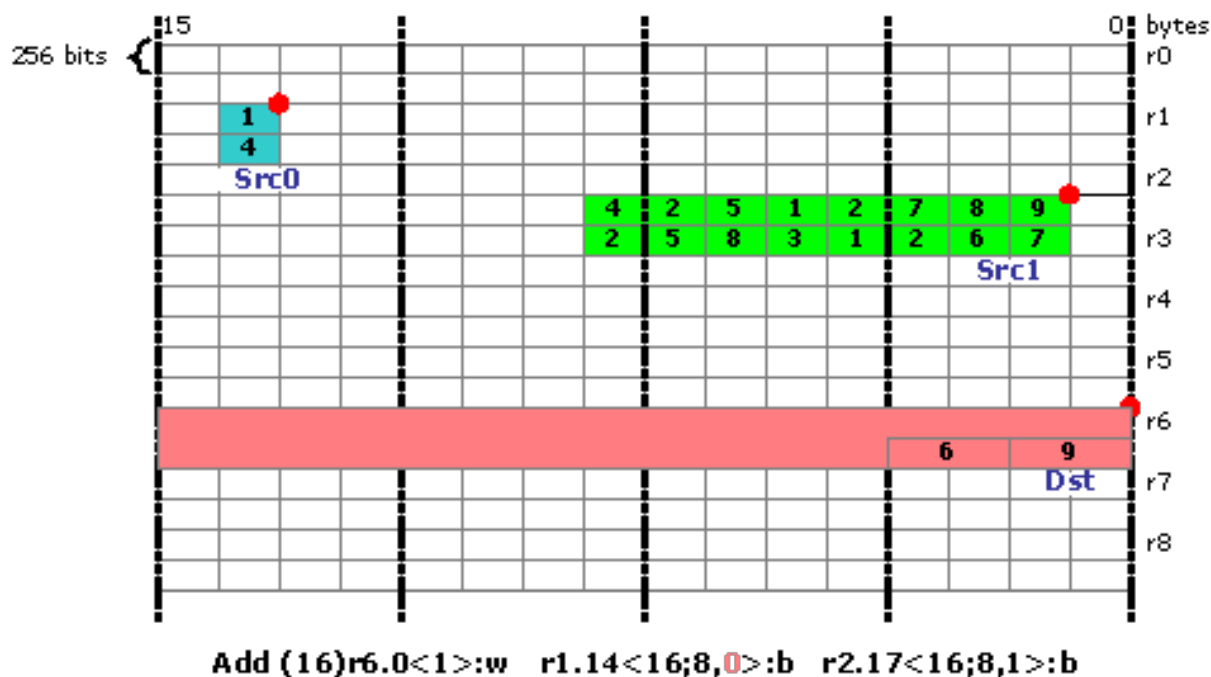
A region description example in direct register addressing



B6903-01

For the example in *Direct Register Addressing*, the sizes of areas of *Src0* and *Src1* are the same, but *Src0* contains a vector of replicated scalars. With *HorzStride* = 0 and *Width* = 8, the first row of 8 elements in *Src0* is a replication of the byte at r1.14. Comparing *ExecSize* of 16 to *Width* of 8 indicates that there is a second row of 8 elements in *Src0*. With *VertStride* = 16, the second row in *Src0* is a replication of the byte at r1.20 (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



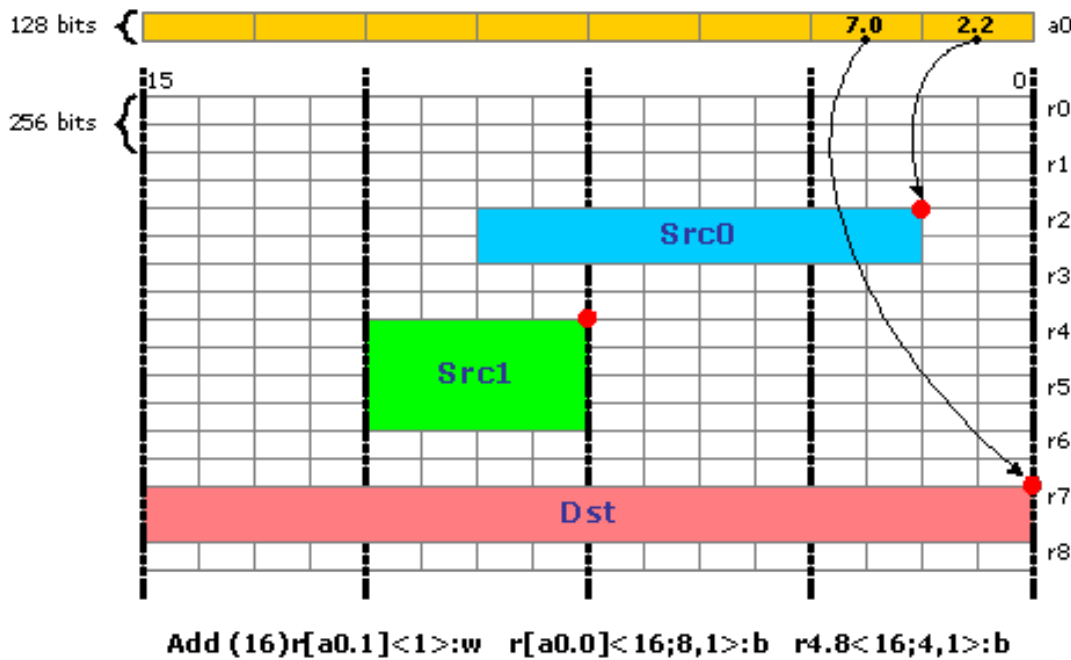
B6904-01

Register-Indirect Register Addressing with a 1x1 Index Region

In the register-indirect register addressing mode with 1x1 index region, the region origin is provided by the content of the address register, the rest of region parameters are provided by the fields in the instruction word.

Register-Indirect Register Addressing with a 1x1 Index Region depicts an example for this addressing mode. For example, the presence of a full region description $\langle 16; 8, 1 \rangle$ 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



B6905-01

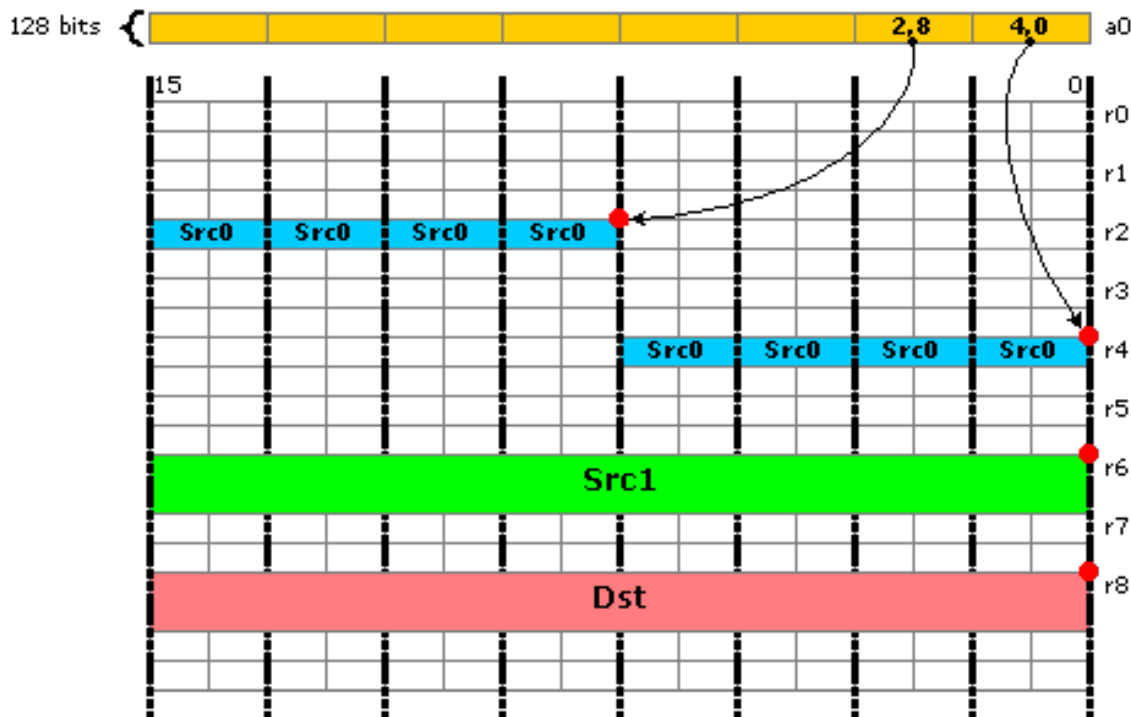
Register-Indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, the horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided and the following contiguous address registers are for the following rows. The total number of address registers used is inferred from the parameters *ExecSize* and *Width*.

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

An example is provided in *Register-Indirect Register Addressing with a Vx1 Index Region*. The assembly syntax notion of a register region without vertical stride, $\langle 4,1 \rangle$, 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 $\text{ExecSize}/\text{Width} = 2$, 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)



Add (8)r8.0<1>:f r[a0.0]<4,1>:w r6.0<4;4,1>:f

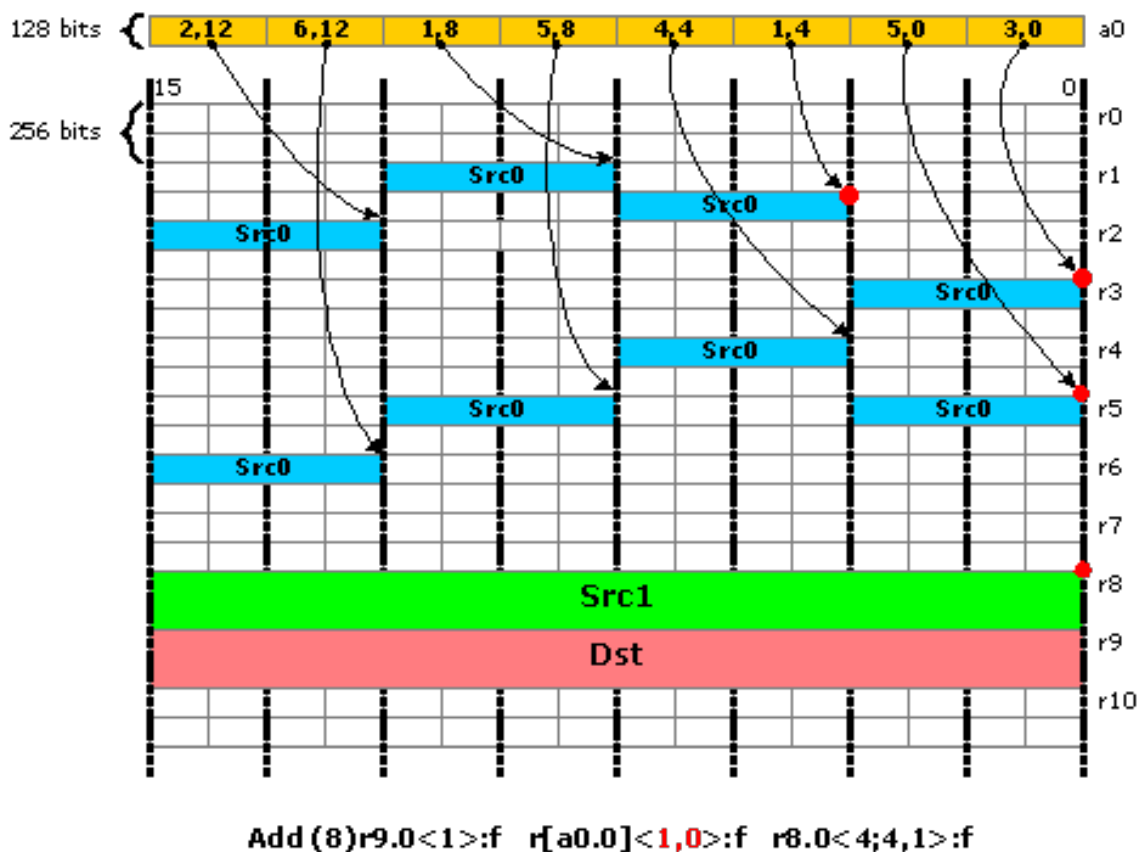
B.6906-01

Register-Indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When *Width* of the region is 1, the number of address registers used equals *ExecSize*.

An example is provided in *Register-Indirect Register Addressing with a VxH Index Region*. The absent of vertical stride in the region description $\langle 1,0 \rangle$ with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As $\text{ExecSize}/\text{Width} = 8$, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.

An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



B6907-01

Access Modes

There are two basic 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-byte aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
- The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.
- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region register descriptions are supported, however, source swizzle or destination mask are not supported.
- The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.

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:

- a. Where n is the largest element size in bytes for any source or destination operand type, $ExecSize * n$ must be ≤ 64 .
- b. 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:

- a. *ExecSize* must be greater than or equal to *Width*.
- b. If $ExecSize = Width$ and $HorzStride \neq 0$, *VertStride* must be set to $Width * HorzStride$.
- c. If $ExecSize = Width$ and $HorzStride = 0$, there is no restriction on *VertStride*.
- d. If $Width = 1$, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
- e. If $ExecSize = Width = 1$, both *VertStride* and *HorzStride* must be 0.
- f. If $VertStride = HorzStride = 0$, *Width* must be 1 regardless of the value of *ExecSize*.
- g. *Dst.HorzStride* must not be 0.
- h. *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

- a. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.
- b. A destination cannot span more than 2 adjacent GRF registers.
- c. 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 PRM limited to other particular projects, the table may appear with no data rows.

Source or Destination Spans Two Registers

Restriction
<p>When an instruction has a source region spanning two registers and a destination region contained in one register, the number of elements must be the same between two sources and one of the following must be true:</p> <ol style="list-style-type: none"> 1. The destination region is entirely contained in the lower OWord of a register. 2. The destination region is entirely contained in the upper OWord of a register. 3. The destination elements are evenly split between the two OWords of a register.
<p>When destination spans two registers, the source may be one or two registers. The destination elements must be evenly split between the two registers.</p> <pre>// Case (a) Two destination registers with one source registers. mov (8) r10.4<1>:f r11.5<8;8,1>:w // Case (b) Two destination registers with two source registers. mov (8) r10.4<1>:f r11.12<8;8,1>:w</pre>

4. Special Cases for Byte Operations

- 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.
- This relaxed alignment rule does not apply for this stepping.

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 PRM limited to other particular projects, the table may not show and there are no special cases in this category.

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 PRM limited to other particular projects, the table may appear with no data rows.

Special Requirements for Handling Double Precision Data Types

Requirement
<p>In Align1 mode, all regioning parameters must use the syntax of a pair of packed floats, including channel selects and channel enables.</p> <pre>// Example: mov (8) r10.0.xyzw:df r11.0.xyzw:df // The above instruction moves four double floats. The .x picks the // low 32 bits and the .y picks the high 32 bits of the double float.</pre>
<p>In Align1 mode, all regioning parameters like stride, execution size, and width are in units of element size. However in Align16 mode, the channel selects and channel enables must always be used in pairs of packed floats, because these parameters are defined for DWord elements ONLY.</p> <pre>// Example: mov (4) r10.0<1>:df r11.0<4;4,1>:df // The above instruction moves four double floats.</pre>
<p>If Align16 is required for an operation with QW destination and non-QW source datatypes, the execution size cannot exceed 2.</p>
<p>In Align16 mode, format conversion from double-float to floats is not allowed when source is immediate data</p>

7. Special Requirements for Handling Mixed Mode Float Operations

There are some special requirements for handling mixed mode float operations for specific projects, described in the following table. If you are viewing a version of the PRM limited to other particular projects, the table may appear with no data rows.

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 PRM limited to other particular projects, the table may appear with no data rows.

Rules
<p>a. When the execution size and destination regioning parameters require two adjacent registers, these registers are accessed using one index register ONLY.</p> <pre>// Example: mov (16) r[a0.0]:f r10:f // The above instruction behaves the same as the following two instructions: mov (8) r[a0.0]:f r10:f mov (8) r[a0.0, 8*4]:f r11:f</pre>

Rules

- b. 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.
```
- c. Indirect addressing on src1 must be a 1x1 indexed region mode.
- d. 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.
```
- e. Indirect addressing on the destination must be a 1x1 indexed region mode.

Execution size of 32 is NOT supported in Vx1 or VxH modes.

9. Special Restrictions

There are some special restrictions on register region access for specific projects, described in the following table. If you are viewing a version of the PRM limited to other particular projects, the table may appear with no data rows.

Restriction
<p>In Align16 mode, the channel selects and channel enables apply to a pair of half-floats, because these parameters are defined for DWord elements ONLY. This is applicable when both source and destination are half-floats.</p> <pre>// Example: mad (8) r10.0.xy:hf r11.0.xxxx:hf r12.0.xyzw:hf r13.xyzw:hf // The .xxxx on the channel select implies the pair of half-floats, r11.0 and r11.1 is replicated. // The .xy on the channel enable implies that 2 DWs or 4 half-floats r10.0, r10.1, r10.2 and r10.3 are updated.</pre>
Ternary instruction with condition modifiers must not use SIMD32.
All flow control (branching) instructions must use the Align1 access mode.
<p>When destination regioning spans two general registers and execution size is less than 8:</p> <ol style="list-style-type: none"> 1. The flag registers cannot be used for implicit update (by condition modifiers). 2. The flag registers cannot be used in a select operation (opcode SEL).
A POW/FDIV operation must not be followed by an instruction that requires two destination registers.
When using Align16 mode for conversion of data elements of different sizes, both source and destination must be one register each.
<p>In Align16 mode, each destination register gets all data from one source register. This means, the data for one destination register is never scattered across two source registers.</p> <pre>// Example: // Allowed - all sources are contained within one register. mul (8) r10.0:f r11.0:f r12.4<0>:f // NOT Allowed - src1 (r14) is scattered across two registers. mad (8) r10.0:f r12.0<0>:f r14.4:f r16.0:f</pre>
<p>Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.</p> <pre>// Example: add (8) r10.0<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.</pre>
<p>Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.</p> <pre>// Example: add (8) r10.0<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.</pre>
When Source is vector immediate, destination must NOT be a byte datatype.
When destination spans two registers, if the execution size is equal to width, horizontal stride must NOT be 0.

Restriction
<p>Any instruction with indirect addressing using vx1 or vxh must use {NoDDCLr} followed by a smov instruction with a {NoDDChk}. The smov instruction writes to the same destination address as the previous instruction and uses a 0x0:ud as immediate</p> <p>// Example:</p> <pre>mov(8) r20:ud r[a0.0]<1,0>:ud {NoDDCLr} smov(1) r20:ud r71:ud 0x00000000:ud {NoDDChk}</pre>
<p>The src, dst overlapping behavior with the second half src and the first half destination to the same register must not be used with any compressed instruction.</p>

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 PRM discusses SIMD execution, both with and without predication. See the subtopics for more details.

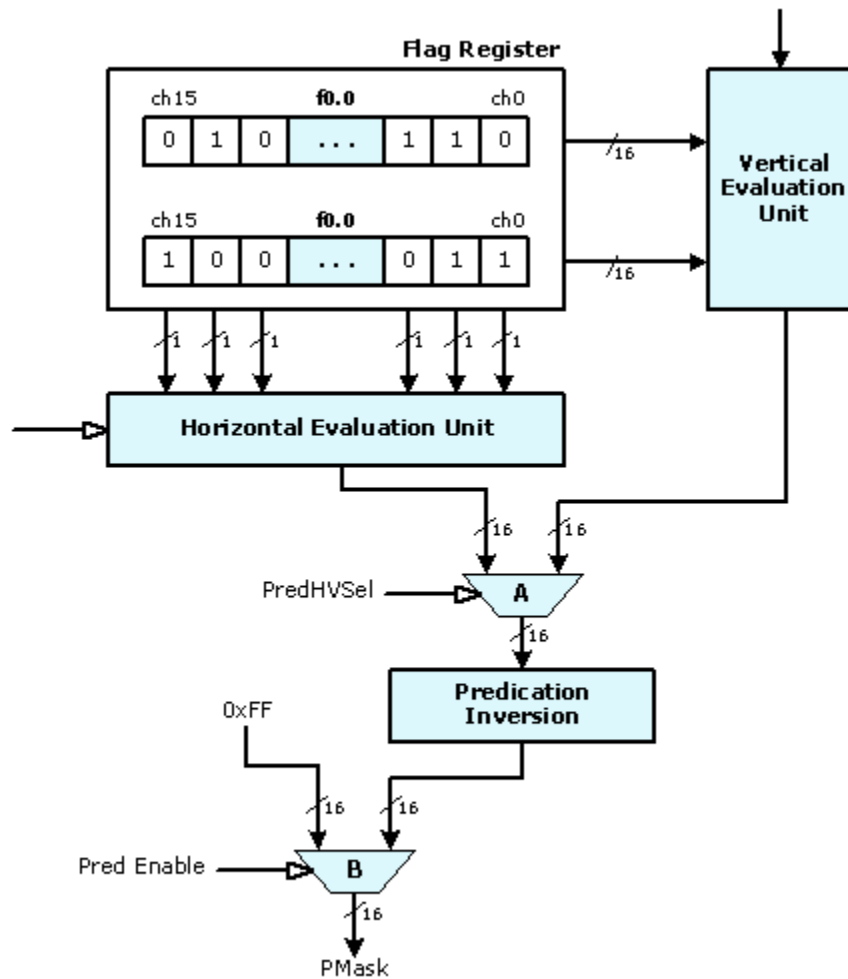
Predication

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 Inversion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.

Generation of predication mask



B6908-01

No Predication

When **PredCtrl** field of a given instruction is set to 0 ("no predication"), it indicates that no predication is applied to this instruction. Effectively, the resulting **PMask** is all 1's. This is shown by the 2:1 multiplexer B controlled by the **Pred Enable** signal in *Predication*. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to **PMask**.

Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is 'no combination' – the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```

If (PredCtrl == "Sequential flag channel mapping") {
    For (ch=0; ch<16; ch++)
        PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}

```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of '.x', '.y', '.z' and '.w' select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = '.x' means that channel 0 in each group is replicated.
- OR combination: PredCtrl of '.any4h' means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of '.all4h' means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```

If (Access Mode == Align16) {
    For (ch = 0; ch < 16; ch += 4)
        Switch (PredCtrl) {
            Case '.x': bTmp = f0.#[ch]; break;
            Case '.y': bTmp = f0.#[ch+1]; break;
            Case '.z': bTmp = f0.#[ch+2]; break;
            Case '.w': bTmp = f0.#[ch+3]; break;
            Case '.any4h': bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
            Case '.all4h': bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
        }
}

```

```

        bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
        PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
    }
}

```

In **Align1** access mode, horizontal combination is based on AND combination '*.any#h*' and OR combination '*.all#h*' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.

```

If (Access Mode == Align1) {
    Switch (PredCtrl) {
        Case '.any2h': groupSize = 2; <op> = '|'; break;
        Case '.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]  
        }  
    }
```

End of Thread

There is no special instruction opcode (such as an END instruction) to cause the thread to terminate execution. Instead, the end of thread is signified by a *send* instruction with the end-of-thread (EOT) sideband bit set. Upon executing a *send* instruction with EOT set, the EU stops on the thread. Upon observing an EOT signal on the output message bus, the Thread Dispatcher makes the thread's resource available. If a thread uses pre-allocated resource managed by a fixed function, such as URB handles and scratch memory, some fixed function protocol also requires the thread to terminate with the message header phase to carry the information in order for the fixed function to release the pre-allocated resource.

EU hardware guarantees that if a terminated thread has in-flight read messages or loads at the time of 'end' that their writebacks will not interfere with either other threads in the system or new threads loaded in the system in the future.

More details can be found in the *send* instruction description in Instruction Reference chapter.

Assigning Conditional Flags

Instructions can output two sets of conditional signals, one set from before the outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. The second set is generated from the final results after clamping and re-normalizing/format conversion logic, and we call this the post conditional signals. The post conditional signals are used for fusing the DirectX compare instruction. **Note:** The flags generated from the post conditional signals should be equivalent to the flags generated by a separate *cmp* instruction after the current arithmetic instruction.

The pre conditional signals are used to generated flags for *cmp/cmpn* instructions only, this logically does the compare of the two input sources. The post conditional signals are used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

cmpn with both sources as NaNs is a don't care case as this doesn't impact the MIN/MAX operations.

The pre conditional signals include the following:

- **pre_sign** bit: This bit reflects the sign of the computed result before going through any kind of clamping, normalizing, or format conversion logic.
- **pre_zero** bit: This bit reflects whether the computed result is zero before any kind of clamping, normalizing, or format conversion logic.

The post conditional signals include the following:

- **post_sign** bit: This bit reflects the sign of the final result after all the clamping, normalizing, or format conversion logic.
- **post_zero** bit: This bit reflects whether the final result is zero after all the clamping, normalizing, or format conversion logic.
- **OF** bit: This bit reflects whether an overflow occurred in any of the computation of the current instruction, including clamping, re-normalizing, and format conversion.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **NS0** bit: The NaN Source 0 bit indicates whether src0 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **NS1** bit: The NaN Source 1 bit indicates whether src1 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0. For an operation with one source operand, this bit is also set to 0. This bit is only used for the comparison instruction *cmpn*, which is specifically provided to emulate MIN/MAX operations. For any other instructions, this bit is undefined.
- Note that the bits generated at the output of a compute are before the **.sat**.

Flag Generation for *cmp* Instructions (The Supported Conditional Modifiers are .e, .ne, .g, .ge, .l, and .le.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
.e	Equal-to	(pre_zero & ! (NS0 NS1)). This conditional modifier tests whether the two sources are equal. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	! (pre_zero & ! (NS0 NS1)). This conditional modifier test whether the two sources are equal. It takes exactly the reverse polarity as the modifier .e .
.g	Greater-than	(! pre_sign & ! pre_zero & ! (NS0 NS1)). This conditional modifier tests whether src0 is greater than src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.ge	Greater-than-or-equal-to	((! pre_sign pre_zero) & ! (NS0 NS1)). This conditional modifier tests whether src0 is greater than or equal to src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.l	Less-than	(pre_sign & !pre_zero & ! (NS0 NS1)). This conditional modifier tests whether src0 is less than src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.le	Less-than-or-equal-to	((pre_sign pre_zero) & ! (NS0 NS1)). This conditional modifier tests whether src0 is less than or equal to src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.

Flag Generation for All Instructions Other than *cmp/cmpn* Instructions (The Supported Conditional Modifiers are *.e*, *.ne*, *.g*, *.ge*, *.l*, *.le*, *.o*, and *.u*.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
.e	Equal-to	(post_zero & ! NC) . This conditional modifier tests whether the result is equal to zero. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	! (post_zero & ! NC) . This conditional modifier test whether the result is not equal to zero. It takes exactly the reverse polarity as modifier .e .
.g	Greater-than	(! post_sign & ! post_zero & ! NC) . This conditional modifier tests whether result is greater than zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.ge	Greater-than-or-equal-to	((! post_sign post_zero) & ! NC) . This conditional modifier tests whether result is greater than or equal to zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.l	Less-than	(post_sign & ! post_zero & ! NC) . This conditional modifier tests whether result is equal to zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.le	Less-than-or-equal-to	((post_sign post_zero) & ! NC) . This conditional modifier tests whether result is equal to or less than zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.o	Overflow	OF . This conditional modifier tests whether the computed result causes overflow – the computed result is outside the range of the destination data type. Note: The legacy condition modifier behavior is different from IEEE exception Overflow flag. For inf float to int conversion, .o will set the legacy Overflow flag, but IEEE exception Overflow flag won't be set. All other internal conditional signals are ignored.
.u	Unordered	NC . This conditional modifier tests whether the computed result is a NaN (unordered). All other internal conditional signals are ignored.

Destination Hazard

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
<p>Instructions other than <i>send</i>, may use this control as long as operations that have different pipeline latencies are not mixed. The operations that have longer latencies are:</p> <ul style="list-style-type: none"> ▪ Opcodes <i>pln, lrp, dp*</i>. ▪ Operations involving double precision computation. ▪ Integer DW multiplication where both source operands are DWs.

When a sequence of *NoDDChk* and *NoDDClr* are used, the last instruction that completes the scoreboard clear must have a non-zero execution mask. This means, if any kind of predication can change the execution mask or channel enable of the last instruction, the optimization must be avoided. This is to avoid instructions being shot down the pipeline when no writes are required.

Example:

```
(f0.0) mov r10.0 r11.0 {NoDDClr}
```

```
(-f0.0) mov r10.0 r11.0 {NoDDChk, NoDDClr}
```

In the above case, if predication can disable all writes to r10 for the second instructions, the instruction maybe shot down the pipeline resulting in un-deterministic behavior. Hence, This optimization must not be used in these cases.

Non-present Operands

Some instructions do not have two source operands and one destination operand. If an operand is not present for an instruction the operand field in the binary instruction must be filled with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand src0. In this case, the second source operand src1 must be set to null and also with the same type as the first source operand src0. It is a special case when src0 is an immediate, as an immediate src0 uses DW3 of the instruction word, which is normally used by src1. In this case, src1 must be programmed with register file ARF and the same data type as src0.

Instruction Prefetch

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 instructions (128 bytes) 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 section contains these sections that introduce this volume.

- [Introducing the Execution Unit](#)
- [EU Terms and Acronyms](#)
- [EU Changes by Processor Generation](#)
- [EU Notation](#)

Subsequent sections cover:

- [EU Data Types](#)
- [Execution Environment](#)
- [Exceptions](#)
- [Instruction Set Summary](#)
- [Instruction Set Reference](#)
- [EU Programming Guide](#)

The EU Programming Guide provides some useful examples and information but is not a complete or comprehensive programming guide.

Introducing the Execution Unit

This section introduces the Execution Unit (EU), a simple and capable processor within the GPU that supports graphics processing within the graphics pipelines, can do general purpose computing (GPGPU), and responds to exceptional conditions via the System Routine.

The EU provides parallelism at two levels: thread and data element. Multiple threads can execute on the EU; the number executing concurrently depends on the processor and is transparent to EU code. Each thread has its own registers (GRF and ARF, described below). Most EU instructions operate on arrays of data elements; the number of data elements is normally the *ExecSize* (*execution size*) or number of channels for the instruction. A *channel* is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels is independent of the number of physical ALUs or FPU's for a particular graphics processor.

EU native instructions are 128 bits (16 bytes) wide. Some combinations of instruction options can use compact instruction formats that are 64 bits (8 bytes) wide. Identifying instructions that can be compacted and creating the compact representations is done by software tools, including compilers and assemblers.

Data manipulation instructions have a destination operand (dst) and one, two, or three source operands (src0, src1, or src2). The instruction opcode determines the number of source operands. An instruction's last source operand can be an immediate value rather than a register.

Data read or written by a thread is generally in the thread's *GRF* (*General Register File*), 128 general registers, each 32 bytes. A data element address within the GRF is denoted by a register number (r0 to r127) and a subregister number. In the instruction syntax, subregister numbers are in units of data element size. For example, a :d (Signed Doubleword Integer) element can be in subregister 0 to 7, corresponding to byte numbers in the instruction encoding of 0, 4, ... 28.

The EU cannot directly read or write data in system memory.

Specialized registers used to implement the ISA are in a distinct per thread *Architecture Register File* (ARF). Each such register or group of related registers has its own distinct name. For example, ip is the instruction pointer and f0 is a flags register. An ARF register can be a src0 or dst operand but not a src1 or src2 operand. There are restrictions on how particular ARF registers are accessed that should be understood before directly reading or writing those registers. See the [ARF Registers](#) section for more information.

The EU supports both integer and floating-point data types, as described in the [Numeric Data Types](#) section.

For EU flow control, each channel has its own per-channel instruction pointer (PcIP[n]) and only executes an instruction when $IP == PcIP[n]$ and any other masks enable the channel. Most flow control instructions use signed offsets from the current instruction address to reference their targets. Unconditional branches are done using *mov* with IP as the destination. Flow control can also use SPF (Single Program Flow) mode to execute with a single instruction pointer (IP).

The EU ISA supports predication, masking, regioning, swizzling, some type conversions, source modification, saturation, accumulator updates, and flag updates as part of instruction execution:

- *Predication* creates a bit mask (*PMask*) to enable or disable channels for a particular instruction execution. Pmask is derived from flag register and subregister values using boolean formulas determined by the PredCtrl (Predicate Control) and PredInv (Predicate Inversion) instruction fields. See the [Predication](#) section.
- *Masking* is the overall process of determining which channels execute for a given instruction based on five factors:
 - Number of channels (only channels in $[0, \text{ExecSize} - 1]$ can execute)
 - Execution mask (*EMask*)
 - Whether the channel is on the instruction (if not in Single Program Flow mode and MaskCtrl is not NoMask)
 - Predicate mask (*PMask*)
 - In Align16 mode, any enabling of channels using the Dst.ChanEn instruction field (if MaskCtrl is not NoMask).
- *Regioning* specifies an array of data elements contained in one or two registers, with options for scattering, interleaving, or repeating data elements in registers using width and stride values, subject to significant constraints. Regioning also includes access mode (Align1 or Align16) and addressing mode (Direct or Indirect). See the [Registers and Register Regions](#) section.
- *Swizzling* allows small scale reordering of data elements within groups of four at the input using the modulo 4 channel names x, y, z, and w. For example, a swizzle of .wzyx with an *ExecSize* of 8 reads execution channels 0 to 7 from these input channels: 3, 2, 1, 0, 7, 6, 5, and 4. Swizzling is only available in the Align16 access mode, described in the Execution Environment chapter.
- *Type Conversions* do any needed conversion from source data type to execution data type and from execution data type to destination data type. See [Execution Data Type](#) for more information. Each instruction description indicates what combinations of data types are supported.
- *Source Modification* modifies a source operand just before doing the requested operation. For a numeric operation, the choices are:
 - No modification (normal).
 - - indicating negation.
 - (abs) indicating absolute value.
 - -(abs) indicating a forced negative value.

Source modification logically occurs after any conversion from source data type to execution data type. Each instruction description indicates whether it supports source modification.

- *Saturation* clamps result values to the nearest value within a saturation range determined by the destination type. For a floating-point type, the saturation range is $[0.0, 1.0]$. For an integer type, the saturation range is the entire range for that type, for example $[0, 65535]$ for the UW (Unsigned Word) type. Each instruction description indicates whether it supports saturation.
- *Accumulator Updates* optionally update the accumulator register or registers in the ARF with destination values as a side effect of instruction execution. The AccWrCtrl instruction field enables accumulator updates. The Accumulator Disable flag in control register 0 (cr0) can be used to

disable accumulator updates, regardless of AccWrCtrl values; for example, this flag may be used in the System Routine.

- *Flag Updates* optionally update a flags register and subregister (f0.0, f0.1, f1.0, or f1.1) with conditional flags based on the CondModifier (Condition Modifier) instruction field. For example, a CondModifier of **.nz** (not zero) assigns flag bits based on whether result elements are not zero (1) or zero (0). Each instruction description indicates whether it supports the Condition Modifier and any restrictions on the values supported.

The EU is not required to execute steps in its internal pipeline sequentially or in order, so long as it produces correct results.

The assembler syntax uses spaces between operands and encloses ExecSize and any predicate in parentheses. Instruction mnemonics, register names, conditional modifiers, predicate controls, and type designators use lowercase. Function names used with the math instruction are UPPERCASE.

```
( pred ) inst cmod sat ( exec_size ) dst src0 src1 { inst_opt, ... }
```

General register destination regions use the syntax *rm.n<HorzStride>:type*. General register directly addressed source regions use the syntax *rm.n<VertStride;Width,HorzStride>:type*. You need to understand more about register regioning to understand all of these terms.

The following example assembly language instruction adds two packed 16-element single-precision Float arrays in r4/r5 and r2/r3 writing results to r0/r1, only on those channels enabled by the predicate in f0.0 along with any other applicable masks.

```
(f0.0) add (16) r0.0<1>:f r2.0<8;8,1>:f r4.0<8;8,1>:f
```

EU Terms and Acronyms

This section provides three tables describing EU general terms and acronyms, EU data types, and EU selected ARF registers.

EU General Terms and Acronyms

Term	Description
ALT mode	A floating-point execution mode that maps +/- inf to +/- fmax, +/- denorm to +/-0, and NaN to +0 at the FPU inputs and never produces infinities, denormals, or NaN values as outputs. See IEEE mode.
ALU	Arithmetic Logic Unit. A functional block that performs integer arithmetic and logic operations, as distinct from instruction fetch and decode, floating-point operations (see FPU), or messaging.
AOS	Array Of Structures. Also see SOA .
ARF	Architecture Register File, a distinct register file containing registers used to implement specific ISA features. For example the Instruction Pointer and condition flags are in ARF registers. See GRF.
byte	An 8-bit value aligned on an 8-bit boundary and the basic unit of addressing. Bits within a byte are denoted 0 to 7 from LSB to MSB.
channel	A logical unit of SIMD data parallel execution within a thread and within the EU. The number of physical ALUs or FPUs is not directly related to the number of channels. BDW supports up to 32 channels.
compact instruction	A 64-bit instruction encoded as described in the EU Compact Instructions section. Only some combinations of instruction parameters can be encoded as compact instructions. See native instruction .
compressed instruction	An instruction that writes to two destination registers. For example a SIMD16 instruction with Float operands can write channels 0 to 7 to one 32-byte general register and channels 8 to 15 to a second, consecutive 32-byte general register.
denorm	A very small but nonzero number in IEEE mode, with a magnitude less than the smallest normalized floating-point number representable in a particular floating-point format. Denormals lose precision as their values approach zero, called <i>gradual underflow</i> .
DWord	Doubleword. A 32-bit (4-byte) value aligned on a 32-bit (4-byte) boundary. Bits within a DWord are denoted 0 to 31 from LSB to MSB.
EOT	End of Thread. A flag set on a <i>send</i> or <i>sendc</i> instruction to terminate a thread's execution on the EU.
EU	Execution Unit. The single GPU unit described in this volume. This volume describes individual data parallel execution paths within a thread in the EU as <i>channels</i> . A few fields, like EUID, use EU to refer to a particular hardware resource used to implement the overall EU.
exception	An error or interrupt condition that arises during execution that may transfer control to the System Routine. Some exceptions can be disabled, preventing such transfers. As defined in this volume, some errors do not produce exceptions.
ExecSize	The number of execution channels for a particular instruction. Channels within that number are enabled or disabled by various masks.

Term	Description
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.
GRF	General Register File, a distinct register file containing 128 general registers, r0 to r127. Each general register is 256 bits (32 bytes), can contain any type of data, and can be accessed with any valid combination of addressing mode, access mode, and region parameters. A general register is directly addressed using a register number and subregister number, or indirectly addressed using an address subregister (index register) and an address immediate offset.
IEEE mode	A floating-point execution mode that supports all the kinds of floating-point values described by the IEEE 754 standard: normalized finite nonzero binary floating-point numbers, signed zeros, signed infinities, signed denormals that are closer to zero than any normalized value but still nonzero, and NaN (not a number) values. See ALT mode.
index register	An address subregister when used for indirect addressing.
inf	Infinity, +inf or -inf, as a floating-point value in IEEE mode.
instruction	In this volume, <i>instruction</i> always refers to an EU instruction.
ISA	Instruction Set Architecture, processor aspects visible to programs and programmers and independent of a particular implementation, including data types, registers, memory access, addressing modes, exceptions, instruction encodings, and the instruction set itself. An ISA does not include instruction timing, hardware pipeline details, or the number of physical resources (ALUs, FPU, 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 <i>packed</i> if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist. The immediate vector data types are all described as <i>Packed</i> because each such type packs several small data elements into a 32-bit immediate value.

Term	Description
QWord	Quadword. A 64-bit (8-byte) value aligned on a 64-bit (8-byte) boundary. Bits within a QWord are denoted 0 to 63 from LSB to MSB.
region	A collection of data locations in registers and subregisters for a source or destination operand. The associated regioning parameters allow regions to be arrays with various layouts.
register	Part of the directly accessible state of an EU program, such as a general register in the GRF or an architecture register in the ARF. Note that system memory is not directly accessible.
SIMD	Single Instruction Multiple Data. Each EU instruction can operate on multiple data elements in parallel, as specified by the instruction's ExecSize.
SIP	System Instruction Pointer, the starting IP value for the System Routine.
SOA	Structure of Arrays. Also see AOS .
SPF	Single Program Flow. A mode in which every execution channel uses the common instruction pointer, IP in the ip register. The SPF bit in the control register is 1 to enable SPF and 0 to disable it. If SPF is disabled, then each execution channel n has its own instruction pointer, PcIP[n] and each channel n is only eligible to execute, subject to other masking, when PcIP[n] == IP.
swizzle	Rearrange data elements within a vector. The EU supports modulo four swizzling of register source operands at the input in the Align16 access mode.
System Routine	A global EU exception handling routine. Any enabled exception from any EU thread transfers control to this routine.
thread	An instance of a program executing on the EU. The life cycle for a thread on the EU starts with the first instruction after being dispatched to the EU by the Thread Dispatcher and ends after executing a <i>send</i> or <i>sendc</i> instruction with EOT set, signaling thread termination. Threads can be independent or can communicate with each other via the Message Gateway shared function.
word	A 16-bit (2-byte) value aligned on a 16-bit (2-byte) boundary. Bits within a word are denoted 0 to 15 from LSB to MSB. <i>Word</i> has denoted a 16-bit unit for Intel processors since the 8086 and 8088 processors were introduced in 1978.

The next table lists all EU numeric data types. See the [Numeric Data Types](#) section for more information about each data type.

EU Numeric Data Types (Listed Alphabetically by Short Name)

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
B	:b	Signed Byte Integer	1	8	I	Signed integer in the range -128 to 127.
D	:d	Signed Doubleword Integer	4	32	I	Signed integer in the range -2^{31} to $2^{31} - 1$.
DF	:df	Double Float	8	64	F	Double precision floating-point number.
F	:f	Float	4	32	F	Single precision floating-point number.
HF	:hf	Half Float	2	16	F	Half precision floating-point number.
Q	:q	Signed Quadword Integer	8	64	I	Signed integer in the range -2^{63} to $2^{63} - 1$.
UB	:ub	Unsigned Byte Integer	1	8	I	Unsigned integer in the range 0 to 255.
UD	:ud	Unsigned Doubleword Integer	4	32	I	Unsigned integer in the range 0 to $2^{32} - 1$.
UQ	:uq	Unsigned Quadword Integer	8	64	I	Unsigned integer in the range 0 to $2^{64} - 1$.
UV	:uv	Packed Unsigned Half Byte Integer Vector	4	32	I	Eight 4-bit unsigned integer values each in the range 0 to 15. Only used as an immediate value.
UW	:uw	Unsigned Word Integer	2	16	I	Unsigned integer in the range 0 to 65,535.
V	:v	Packed Signed Half Byte Integer Vector	4	32	I	Eight 4-bit signed integer values each in the range -8 to 7. Only used as an immediate value.
VF	:vf	Packed Restricted Float Vector	4	32	F	Four 8-bit restricted float values. Only used as an immediate value.
W	:w	Signed Word Integer	2	16	I	Signed integer in the range -32,768 to 32,767.

The next table lists the seven ARF registers that you should understand first, omitting several others. See the [ARF Registers](#) section for more information, including descriptions of additional registers not listed below.

EU Selected ARF Registers (Listed Alphabetically by Name)

Name	Assembler Syntax	Description
Accumulators	acc0, acc1	Data registers that can hold integer or floating-point values of various sizes. Many instructions can implicitly update accumulators with a copy of destination values, done by setting the AccWrCtrl instruction option. A few instructions, like <i>mac</i> (Multiply Accumulate), use the accumulators as an implicit source operand, useful for some iterative calculations. For BDW 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). For BDW 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	fr.s	Used as the outputs for various channel conditional signals, such as equality/zero or overflow. Used as the inputs for predication. There are two 32-bit flags registers each containing two 16-bit subregisters.
Instruction Pointer (IP)	ip	References the current instruction in memory, as an unsigned offset from the General State Base Address. IP is the thread's overall instruction pointer. Each channel n can have its own instruction pointer (PcIP[n]). If not in Single Program Flow mode (SPF is 0) then only those channels where PcIP[n] == IP are eligible to execute the instruction, if enabled by all other applicable masks.
Null Register	null	Indicates a non-existent operand. Unused operands in the instruction format, like the unused second source operand field in a <i>mov</i> instruction, are encoded as null. For present source operands, reading a null source operand returns undefined values. For null destination operands, results are discarded but any implicit updates to accumulators or flags still occur.
State Register	sr0.s	Contains thread identification and scheduling fields, and mask fields for enabling or disabling channels.

Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on. Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple 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 <i>Message Gateway, Media</i>)
synchronization of thread output to memory buffers (see <i>Geometry Shader</i>)

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs.

EU Changes 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
<p>These features or behaviors are added for BDW:</p> <ul style="list-style-type: none"> • The maximum <i>ExecSize</i> increases to 32, for byte or word operands. • Increase the number of flag registers from one to two. • Add the <i>NibCtrl</i> field, used with <i>QtrCtrl</i> to select groups of channels or flags. • Add the DF (Double Float) data type, the first time an 8-byte data type is supported. DF only supports the IEEE floating-point mode and not the ALT floating-point mode. • Add a shared source data type field and a destination data type field for instructions with three source operands, allowing F (Float), DF (Double Float), D (Signed Doubleword Integer), or UD (Unsigned Doubleword Integer) types to be specified. • Add bit manipulation instructions: <i>bfi1</i>, <i>bfi2</i>, <i>bprev</i>, <i>cbit</i>, <i>fbh</i>, and <i>fbl</i>. • Add the integer <i>addc</i> (Add with Carry) and <i>subb</i> (Subtract with Borrow) instructions. • Add the <i>brc</i> (Branch Converging) and <i>brd</i> (Branch Diverging) instructions. • For the <i>cmp</i> and <i>cmpn</i> instructions, relax the accumulator restrictions. • For the <i>sel</i> instruction, remove the accumulator restriction. • Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0.
<p>These features or behaviors are added for BDW, continuing to later generations:</p> <ul style="list-style-type: none"> • DF (Double Float) operands use an element size of 8. Regioning and channel parameters for the DF type are determined normally, in the same way as for other types. • Add the channel enable register, flow control registers, and stack pointer register in the ARF. • In the Control Register, add the Force Exception Status and Control, Context Save Status, and Context Restore Status bits. • Relative instruction offsets (JIP, UIP) are now 32-bit values in units of bytes (rather than 16-bit values using 8-byte units) for some instructions: <i>brc</i>, <i>brd</i>, <i>call</i>, and <i>jmp</i>. • A <i>call</i> instruction can get the relative instruction offset (JIP) from a register. • Add the <i>calla</i> (Call Absolute) instruction. • A <i>mov</i> instruction with different source and destination types can now use conditional modifiers.
<p>These features or behaviors are added for BDW, continuing to later generations:</p> <ul style="list-style-type: none"> • 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.

Description
<ul style="list-style-type: none"> • Add the Single Precision Denorm Mode bit in Control Register 0. It can be enabled to allow calculations using the F (Float) type in IEEE floating-point mode to support denormals and gradual underflow. • Add the Q (Signed Quadword Integer) and UQ (Unsigned Quadword Integer) types. Integer source types cannot mix 64-bit and 8-bit operands. Some integer instructions (e.g., <i>avg</i>) do not support Q or UQ source types. • Instructions with one source operand and a 64-bit source type can have immediate 64-bit source operands. • The JIP and UIP relative instruction offset fields in all remaining flow control instructions are 32-bit values in units of bytes (rather than 16-bit values using 8-byte units). • The instruction layout is noticeably different. The SrcType and DstType instruction fields are widened to allow for more type encodings as three types are added. The AddrSubRegNum instruction field is widened to allow for 16 address subregisters rather than 8. The layout now supports 64-bit immediate source operands for one-source instructions and 32-bit relative instruction offset fields for flow control instructions. • In the 3-source instruction format, widen the SrcType and DstType fields and add an encoding for the HF (Half Float) type. • Add a compact instruction format for 3-source instructions. • Use a different source modifier interpretation for logical (<i>and</i>, <i>not</i>, <i>or</i>, <i>xor</i>) 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 <i>brc</i> and <i>brd</i> instructions do not allow the Switch instruction option. • For the <i>cmp</i> and <i>cmpn</i> instructions, remove the accumulator restrictions. • Add the <i>goto</i> instruction, reusing the opcode for the discontinued <i>fork</i> instruction. • Add the <i>join</i> instruction. • For the <i>lzd</i> instruction, remove the accumulator restriction. • The <i>mach</i> instruction reverses the roles of the two source operands compared to previous generations. • Add the <i>madm</i> instruction. • Enhance the <i>math</i> instruction to allow some immediate source values and support the INVM and RSQRTM functions. • For the <i>mul</i> instruction, relax the accumulator restriction on source operands so it applies for only integer source operands. • For the rounding instructions (<i>rndd</i>, <i>rnds</i>, <i>rndu</i>, and <i>rndz</i>), remove the accumulator restrictions. • Revise the <i>shl</i> and <i>shr</i> instructions to use the low 6 bits of the shift count in QWord mode, versus the low 5 bits otherwise. • Add the <i>smov</i> 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.

Description
<ul style="list-style-type: none"> Remove some regioning restrictions when operands span two registers.
<p>These features or behaviors are specific to BDW and may not continue to later generations:</p> <ul style="list-style-type: none"> Calculations using the HF (Half Float) type do not support denormals or gradual underflow. Revise the Slice ID and SubSlice ID (previously HalfSlice ID) bit fields in the State Register. The System IP (SIP) is widened to 48 bits. However the EU still only uses the low 32 bits.

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 PRM 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.

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

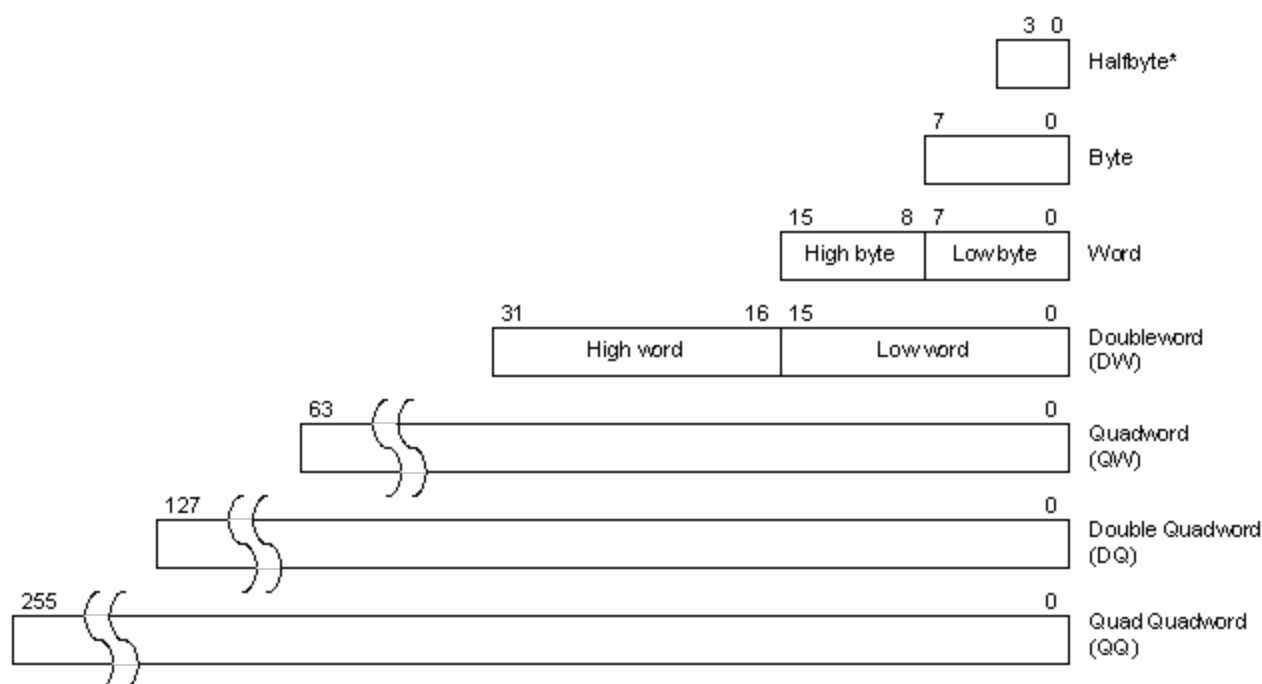
- IEEE Floating-Point Exceptions
- Floating-Point Compare Operations

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.

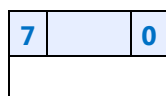
Numeric Data Types

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.

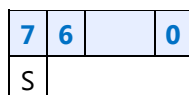
Integer Numeric Data Types

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

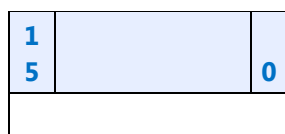
UB: Unsigned Byte, 8-bit Unsigned Integer



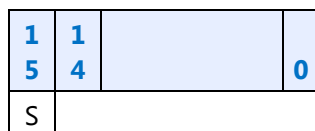
B: Byte, 8-bit Signed Integer



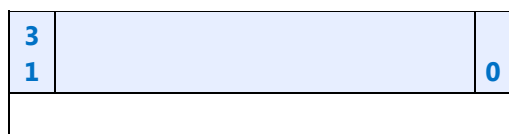
UW: Unsigned Word, 16-bit Unsigned Integer



W: Word, 16-bit Signed Integer



UD: Unsigned Doubleword, 32-bit Unsigned Integer



D: Doubleword, 32-bit Signed Integer



UQ: Unsigned Quadword, 64-bit Unsigned Integer BDW

6 3		0

Q: Quadword, 64-bit Signed Integer BDW

6 3	6 2		0
S			

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

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

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

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

The following table summarizes the EU integer data types.

Execution Unit Integer Data Types

Notation	Size in Bits	Name	Range
UB	8	Unsigned Byte Integer	[0, 255]
B	8	Signed Byte Integer	[-128, 127]
UW	16	Unsigned Word Integer	[0, 65535]
W	16	Signed Word Integer	[-32768, 32767]
UD	32	Unsigned Doubleword Integer	[0, $2^{32} - 1$]
D	32	Signed Doubleword Integer	$[-2^{31}, 2^{31} - 1]$
UQ	64	Unsigned Quadword Integer	[0, $2^{64} - 1$]
Q	64	Signed Quadword Integer	$[-2^{63}, 2^{63} - 1]$
UV	32	Packed Unsigned Half-Byte Integer Vector	[0, 15] in each of eight 4-bit immediate vector elements.
V	32	Packed Signed Half-Byte Integer Vector	[-8, 7] in each of eight 4-bit immediate vector elements.

Restriction: Only a raw move using the *mov* instruction supports a packed byte destination register region. For information about raw moves, refer to the **Description** in *mov – Move*.

Restriction: Q/UQ data types are not supported in Align16 mode.

Floating-Point Numeric Data Types

The Execution Unit supports the following floating-point data types.

Type or Description
The Float type uses the single precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
The Double Float type uses the double precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
In the ALT floating-point mode, representations for infinities, denorms, and NaNs within those formats are not used. The EU does not support the double extended precision (80-bit) floating-point format found in the x86/x87/Intel 64 floating-point registers. All floating-point formats are signed using signed magnitude representation (a distinct sign bit, separate from the magnitude information).
The Half Float type uses the binary16 format specified in IEEE Standard 754-2008.
The F (Float) type supports both the ALT and IEEE floating-point modes, controlled by the Single Precision Floating-Point Mode bit in the Control Register.
Whether IEEE mode F calculations support denorms or flush denormalized values to zero is controlled by the Single Precision Denorm Mode bit in the Control Register.
The DF (Double Float) type only supports the IEEE floating-point mode. Whether DF calculations support denorms or flush denormalized values to zero is controlled by the Double Precision Denorm Mode bit in the Control Register.
The HF (Half Float) type only supports the IEEE floating-point mode.
HF calculations flush denormalized values to zero and gradual underflow is not supported.

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

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

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

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

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

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

VF: Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number

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

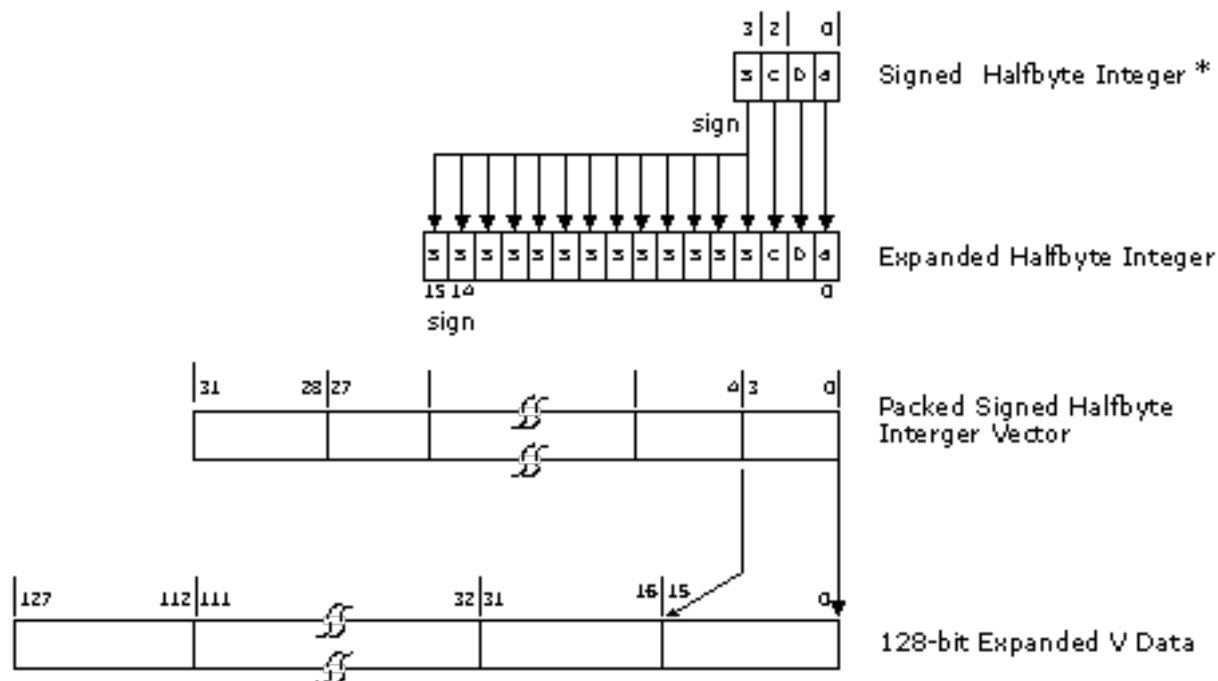
Notation	Size in Bits	Name	Range
HF	16	Half Float	Half precision, 1 sign bit, 5 bits for the biased exponent, and 10 bits for the significand: $[-(2-2^{-10})^{31} \dots -2^{-40}, 0.0, 2^{-40} \dots (2-2^{-10})^{31}]$
F	32	Float	Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand: $[-(2-2^{-23})^{127} \dots -2^{-149}, 0.0, 2^{-149} \dots (2-2^{-23})^{127}]$
DF	64	Double Float	Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: $[-(2-2^{-52})^{1023} \dots -2^{-1074}, 0.0, 2^{-1074} \dots (2-2^{-52})^{1023}]$
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: $[-31 \dots -0.125, 0, 0.125 \dots 31]$

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**.

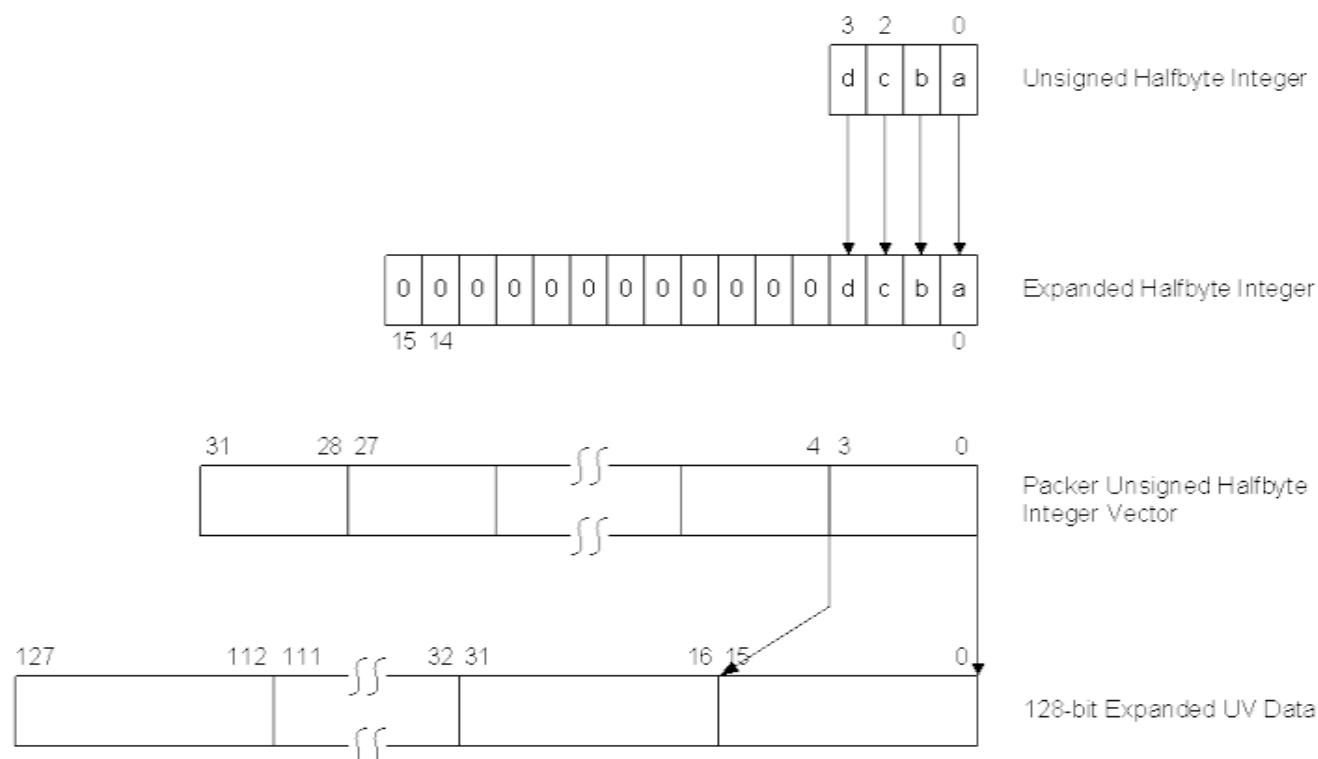
Converting a Packed Half-Byte Vector to a 128-bit Signed Integer Vector



B6885-01

Packed UnSigned Half-Byte Integer Vector

A packed unsigned halfbyte integer vector consists of 8 unsigned halfbyte integers contained in a doubleword. Each unsigned halfbyte integer element has a range from 0 to 15. This numeric data type is only used by an immediate source operand of doubleword in 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.

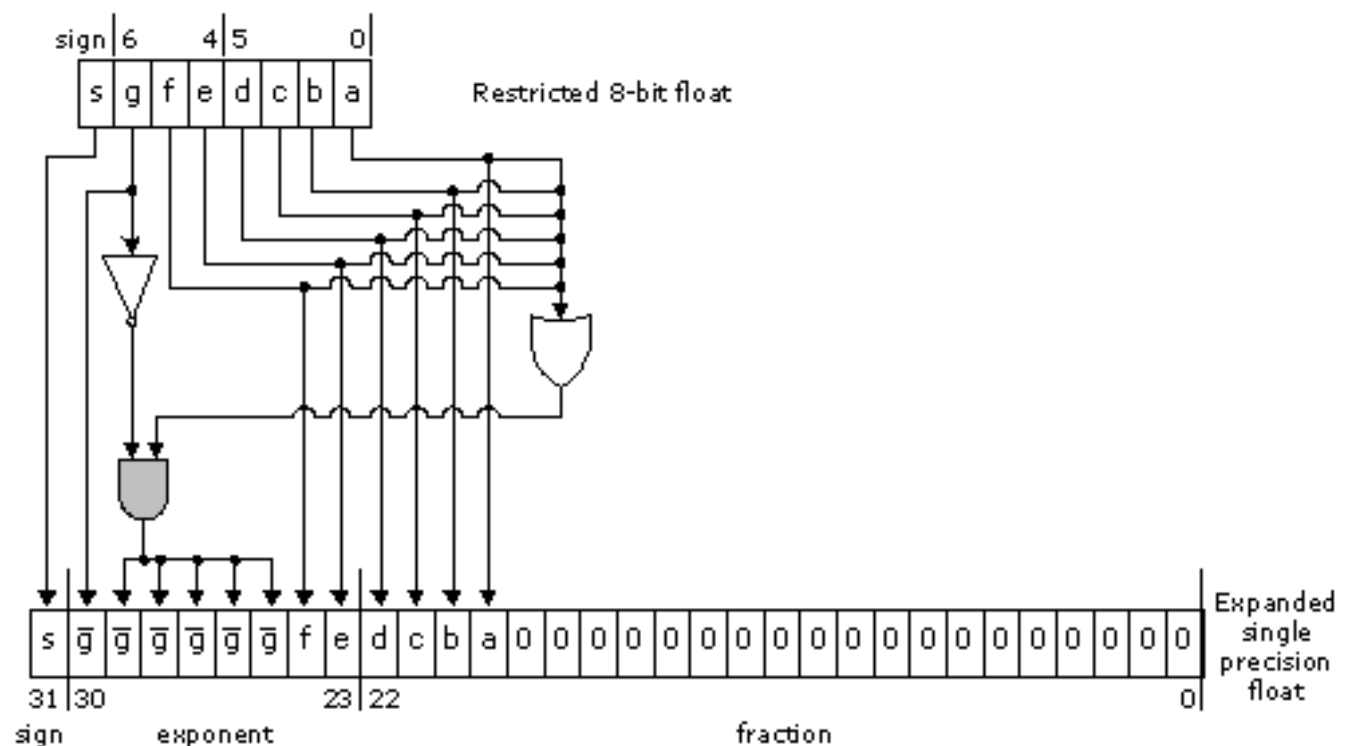


Packed Restricted Float Vector

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



B6886-01

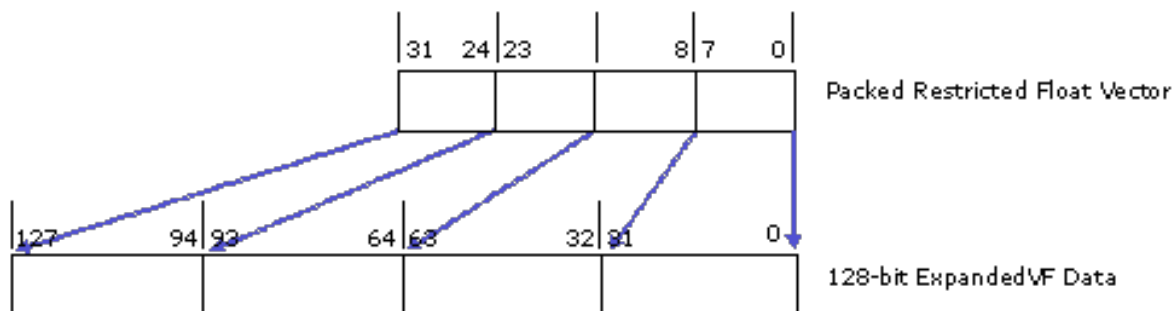
The following table shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinities, NaN, and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range – the magnitude of the smallest non-zero number is 0.125.

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	0.125 ... 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	-0.125 ... -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.



B.6889-01

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.

- $\text{INF} - \text{INF} = \text{NaN}$
- $0 * (+/-)\text{INF} = \text{NaN}$
- $1 / (+\text{INF}) = +0$ and $1 / (-\text{INF}) = -0$
 - $(+/-)\text{INF} / (+/-)\text{INF} = \text{NaN}$ as $A/B = A * (1/B)$
- $\text{INV} (+0) = \text{RSQ} (+0) = +\text{INF}$, $\text{INV} (-0) = \text{RSQ} (-0) = -\text{INF}$, and $\text{SQRT} (-0) = -0$
- $\text{RSQ} (-\text{finite}) = \text{SQRT} (-\text{finite}) = \text{NaN}$
- $\text{LOG} (+0) = \text{LOG} (-0) = -\text{INF}$, $\text{LOG} (-\text{finite}) = \text{LOG} (-\text{INF}) = \text{NaN}$

- 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.

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 $\pm inf$. Thus, compatibility between a fused operation, such as *mac*, with the unfused equivalent, *mul* followed by *add* in this case, is not guaranteed.
- As the accumulator registers have more precision than 32-bit float, any instruction with accumulator as a source/destination operand may produce a different result than that using more general registers, as indicated in this table:
Such an instruction may produce a different result than that using GRF registers.
- API Shader divide operations are implemented as $x * (1.0f/y)$. With the two-step method, $x * (1.0f/y)$, the multiply and the divide each independently operate at the 32-bit floating point precision level (accuracy to 1 ULP).
- See the [Type Conversion](#) section for rules on converting to and from float representations.

Min Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in the 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.

This also implies that ALT mode is not supported when Single precision datatype is involved in format conversion to double precision or half precision.

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: \pm finite (normalized nonzero finite number), ± 0 (signed zero), $\pm \text{inf}$ (signed infinity), \pm denorm (denormalized, very small but nonzero number), or NaN (Not a Number). A NaN can be a signaling NaN (sNaN) or quiet NaN (qNaN).

These operating modes are available for the different floating-point types:

- Half Float uses the IEEE floating-point mode.
Half Float does not support denormals. Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign.
- 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 $\pm f_{\text{max}}$, the largest finite magnitude representable in a format, and $\pm f_{\text{min}}$, the smallest normalized nonzero magnitude representable in a format. Calculating those values uses the extreme exponent values for finite nonzero floating-point values, E_{max} and E_{min}

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	E _{max}	E _{min}	Explicit Fraction Bits	f _{max}	f _{min}
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 f_{max} is $(2.0 - 2^{-f}) * 2^{E_{\max}}$.

The general formula for f_{min} is $2^{E_{\min}}$.

Not a Number (NaN) Formats

A NaN has a biased exponent field with all bits set (as if encoding an exponent of E_{max} + 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 $\pm \text{inf}$ and RZ always produces $\pm \text{fmax}$.

Floating-Point Operations and Precision

IEEE Standard 754-2008 requires the following floating-point operations to be precise within $\leq 0.5 \text{ ulp}$ (unit in the last place) when using the round to nearest or even rounding mode:

- ADD (*add* instruction)
- DIV (*math* instruction with FDIV function code)
- FMA (fused multiply add, *mad* instruction)
- MUL (*mul* instruction)
- SQRT (*math* instruction with SQRT function code)
- SUB (*add* instruction using one - source modifier)
- Conversions (float to float, float to int, and int to float)
- Min/Max
- Compare

Single Precision Floating-Point Rounding to Integral Values

The *rndd* (Round Down), *rnde* (Round to Nearest or Even), *rndu* (Round Up), and the *rndz* (Round to Zero) instructions round arbitrary Float values to integral Float values. Each instruction specifies its rounding mode so these instructions are not affected by the Rounding Mode in the Control Register.

An integral source value produces the same value for the destination (ignoring any saturation). For magnitudes $\geq 8,388,608$ (2^{23}) all Float values are integral.

The rounding instructions are sign preserving.

Signed zeros are propagated. In IEEE mode, signed infinities are propagated. In IEEE mode, sNaN inputs are quietized, the equivalent qNaN is produced, and the Invalid Operation exception is indicated. In IEEE mode, qNaN inputs are propagated.

No other exceptions are signaled for these instructions. For example if the source and result values differ, the Inexact exception is not signaled.

The Single Precision Denorm Mode in the Control Register affects the results of the *rndd* and *rndu* instructions for denorm source values.

Floating-Point to Integer Conversion

The *mov* and *sel* instructions can be used to convert floating-point values to integers. In the tables below, *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
$f > \text{Imax}$	Imax	Invalid Operation
$\text{Imin} \leq f \leq \text{Imax}$	<i>f</i>	Inexact if rounding changed <i>f</i>
$f < \text{Imin}$	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
$f > \text{Imax}$	Imax	Invalid Operation
$0 \leq f \leq \text{Imax}$	<i>f</i>	Inexact if rounding changed <i>f</i>
$f = -0$	0	Inexact
$-1 < f < -0$	0	Inexact
$-1 < f < -0$	Real Indefinite	Invalid Operation
$-\text{fmax} \leq f \leq -1$	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 $\pm \text{inf}$, 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 $\pm \text{fmax}$ or $\pm \text{inf}$, as described in the [Overflow Exception](#) section.

Floating-Point Min/Max Operations

In the following Min/Max operations, sNaN inputs are preferred to non-NaN inputs and non-NaN inputs are preferred to qNaN inputs.

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

$\text{Min}(x, \text{sNaN}) = \text{Min}(\text{sNaN}, x) = \text{qNaN}$ (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

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

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

$\text{Max}(x, \text{sNaN}) = \text{Max}(\text{sNaN}, x) = \text{qNaN}$ (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

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

Special cases(when both sources are NaN inputs)

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

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

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

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

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

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

IEEE Floating-Point Exceptions

The BDW EU detects the five floating-point exceptions defined by IEEE Standard 754:

Invalid Operation

Division by Zero

Overflow

Underflow

Inexact

Description
When converting from float to int, IEEE exception flags are not updated correctly.
Non IEEE compliant instructions updating exception flags
float_to_int conversion is supported in HW only for RTZ mode

Signaling Floating-Point Exceptions

When enabled in the [Control Register](#), floating-point exceptions are detected and set *sticky* flag bits in the [State Register](#). There is no mechanism for automatic transfer to a handler, so floating-point exceptions are not handled like other exceptions described in the [Exceptions](#) chapter.

Setting flags to indicating Floating-Point exceptions is the default exception handling approach specified by IEEE Standard 754. The flag bits are sticky because in normal operation the EU only sets these bits as exceptions occur, and does not clear these bits, so a set value sticks until cleared by software. The Control Register and State Register are cleared at reset and initialized at thread load. Both are read/write registers. These fields are used:

- **IEEE Exception Trap Enable** (Control Register cr0.0:ud bit 9). This bit enables trapping IEEE exception flags. This control bit may be updated by software. It is initially zero on thread load. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exception field of sr0.1, in the State Register. **Note:** Software must set this flag at thread start to use the IEEE Exception flags.
- **IEEE Exception.** (State Register sr0.1:ud bits 7:0). The IEEE exception bits are sticky bits set by the opcodes when floating-point exceptions are triggered. These bits are defined per thread and all channels update one set of sticky bits. These bits are cleared on thread load and may be cleared by software. Exception updates to these bits may be disabled by clearing the IEEE Exception Trap Enable bit in the Control Register. The following table specifies the IEEE exception bits:

Bits	Definition
7:5	Reserved
4	Inexact Exception
3	Overflow Exception
2	Underflow Exception
1	Division by Zero Exception
0	Invalid Operation Exception

The IEEE exception flags are per thread, shared by all channels. (Maintaining separate per channel exception flags for 32 channels would require 160 bits per thread.)

Invalid Operation Exception

An Invalid Operation exception is signaled by any operation on a signaling NaN (sNaN) or by certain combinations of operations and operands with undefined results, always producing a quiet NaN (qNaN) result.

The following specific operations signal Invalid Operation, where x is a positive, finite, nonzero, and normalized number:

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

$\pm 0 / \pm 0$

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

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

$\text{Remainder}(\pm x, \pm 0)$

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

$\text{Sqrt}(-x)$

Note that $\text{Sqrt}(-0)$ 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

smove except for raw moves

Division by Zero Exception

The operation $\pm x / \pm 0$, where x is a positive, finite, nonzero, and normalized number, signals the Division by Zero Exception and produces a correctly signed $\pm \text{inf}$ result. Note that in accordance with the standard, $\pm 0 / \pm 0$ signals Invalid Operation, does not signal Division by Zero, and produces a qNaN result. This behavior can occur for the *math* instruction with the FDIV function code.

The operation $\text{LOG}_2(\pm 0)$ signals the Division by Zero Exception and produces $-\text{inf}$ as the result. This behavior can occur for the *math* instruction with the LOG function code.

Overflow Exception

A floating-point *overflow* occurs when an operation with a finite result produces an internal result with magnitude $> \text{fmax}$, 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 $\pm \text{inf}$ or $\pm \text{fmax}$ 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 ($\text{abs}(\text{rounded unbounded result}) > \text{fmax}(\text{destination format})$) {

Set the Overflow Exception flag to 1.

Output = $\pm \text{inf}$ or $\pm \text{fmax}$ depending on the sign and the rounding mode:

+ and (RNE or RU): $+\text{inf}$

+ and (RD or RZ): $+\text{fmax}$

- and (RU or RZ): $-\text{fmax}$

- and (RNE or RD): $-\text{inf}$

}

}

Note: The first three steps of the overflow and underflow algorithms are identical.

Underflow Exception

An *underflow* occurs when an operation produces a tiny but nonzero inexact result x , with $\text{abs}(x) < f_{\text{min}}$, where $f_{\text{min}} = 2^{E_{\text{min}}}$. See the [Floating-Point Type Parameters](#) table for the f_{min} and E_{min} values for different floating-point types.

IEEE Standard 754 allows underflow to be determined before or after rounding. The Execution Unit determines underflow after rounding, which is consistent with the behavior of the x87 and SSE instructions in the CPU.

When denorms are flushed to zero, no underflow exceptions are signaled. Flush to zero is not defined by IEEE Standard 754, but is implementation specific and required by some APIs (including DirectX), thus the EU ISA 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 f_{min} but nonzero AND the denorm result in the destination format is inexact, then signal the Underflow Exception.

The rounded result can be ± 0 , $\pm f_{\text{min}}$, or (when denorms are enabled) $\pm \text{denorm}$.

IEEE Standard 754 requires the non-intuitive behavior that an exact denorm result does not set the Underflow Exception flag.

The following algorithm describes floating-point underflow processing:

1. Compute the result with infinite precision and unbounded range.
2. Normalize the result using an unbounded exponent.
3. Round the result to the destination precision using an unbounded exponent and the current rounding mode.
4. If ($0 < \text{abs}(\text{rounded unbounded result}) < f_{\text{min}}(\text{destination format})$) {

if (flush denorms to zero) {
 Output = 0; // No underflow exception.
 }
 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;
    }

```

Restriction: Underflow Flag is not supported.

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. $\text{src0} < \text{src1}$ and neither source is NaN.

Equal. $\text{src0} = \text{src1}$ and neither source is NaN.

Greater than. $\text{src0} > \text{src1}$ and neither source is NaN.

Unordered. Any source is NaN.

Any NaN compares unordered to any value, including itself.

Infinities of the same sign compare as equal.

Zeros compare as equal regardless of sign: $-0 = +0$.

Floating-Point Compare Relations

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	E	L	L	L	L	L	L	L	U
-fin	G	*	L	L	L	L	L	L	U
- denorm	G	G	*^	L^	L^	L^	L	L	U
-0	G	G	G^	E	E	L^	L	L	U
+0	G	G	G^	E	E	L^	L	L	U
+denor m	G	G	G^	G^	G^	*^	L	L	U
+fin	G	G	G	G	G	G	*	L	U
+inf	G	G	G	G	G	G	G	E	U
NaN	U	U	U	U	U	U	U	U	U
Notes									
*	Relation can be L, E, or G.								
^	When denorms are flushed to zero then all denorms and zeros compare as E.								

The next six tables show the results of six specific comparisons, corresponding to the **.g**, **.l**, **.e**, **.ne**, **.ge**, and **.le** conditional modifiers used with the *cmp* instruction and a floating-point source type. Any NaN source produces a false comparison result for these modifiers other than **.ne** and produces a true comparison result for the **.ne** modifier.

Results of Greater Than Comparison — *cmp.g*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	F	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
- denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	F	F	F^	F	F	F
+0	T	T	T^	F	F	F^	F	F	F
+denor m	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	F	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the .g comparison result F.								

Results of Less Than Comparison — *cmp.l*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
- denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	F	F	T^	T	T	F
+0	F	F	F^	F	F	T^	T	T	F
+denor m	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	F	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the .l comparison result F.								

Results of Equal Comparison — *cmp.e*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	F	*	F	F	F	F	F	F	F
- denorm	F	F	*^	F^	F^	F^	F	F	F
-0	F	F	F^	T	T	F^	F	F	F
+0	F	F	F^	T	T	F^	F	F	F
+denor m	F	F	F^	F^	F^	*^	F	F	F
+fin	F	F	F	F	F	F	*	F	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the .e comparison result T.								

Results of Not Equal Comparison — *cmp.ne*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	T
-fin	T	*	T	T	T	T	T	T	T
- denorm	T	T	*^	T^	T^	T^	T	T	T
-0	T	T	T^	F	F	T^	T	T	T
+0	T	T	T^	F	F	T^	T	T	T
+denor m	T	T	T^	T^	T^	*^	T	T	T
+fin	T	T	T	T	T	T	*	T	T
+inf	T	T	T	T	T	T	T	F	T
NaN	T	T	T	T	T	T	T	T	T
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the .ne comparison result F.								

Results of Greater Than or Equal Comparison — *cmp.ge*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
- denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	T	T	F^	F	F	F
+0	T	T	T^	T	T	F^	F	F	F
+denor m	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	T	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the .ge comparison result T.								

Results of Less Than or Equal Comparison — *cmp.le*

src1 src0	-inf	-fin	- denorm	-0	+0	+denor m	+fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
- denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	T	T	T^	T	T	F
+0	F	F	F^	T	T	T^	T	T	F
+denor m	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the <i>.le</i> comparison result T.								

Type Conversion

Float to Integer

Converting from float to integer is based on rounding toward zero (RTZ is for DX, IEEE expects all four rounding modes). If the floating point value is +0, -0, +Denorm, -Denorm, +NaN or -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UD, D, UW, W, UB and B).

Input Format	Output Format					
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
-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	<ul style="list-style-type: none"> • A bit in the instruction word • Breakpoint IP match • Breakpoint Opcode match 	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous
Context Save/Restore	Preemption Interrupt	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver, and delivered to the EU as part of a new thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's cr0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception type recognition across hardware threads. The exception enable bits in the cr0.1 register are read/write, and thus can be enabled/disabled via software at any time during thread execution.

The exception handling mechanism relies on the System Routine, a single subroutine that provides common exception handling for all threads on all EUs in the system. This System Routine is defined per-context and is identified via a System IP (SIP) register in context state. At the time of each context switch, the appropriate SIP for that context is loaded into each EU, allowing each context to have custom implementation of exception handling routines if so desired.

The mechanism does not support handling recursive system routine access. This means when a thread cannot be asynchronously interrupted to an exception when executing a SIP.

Example:

An Exception is not supported when hardware is executing a SIP for context save and restore operations.

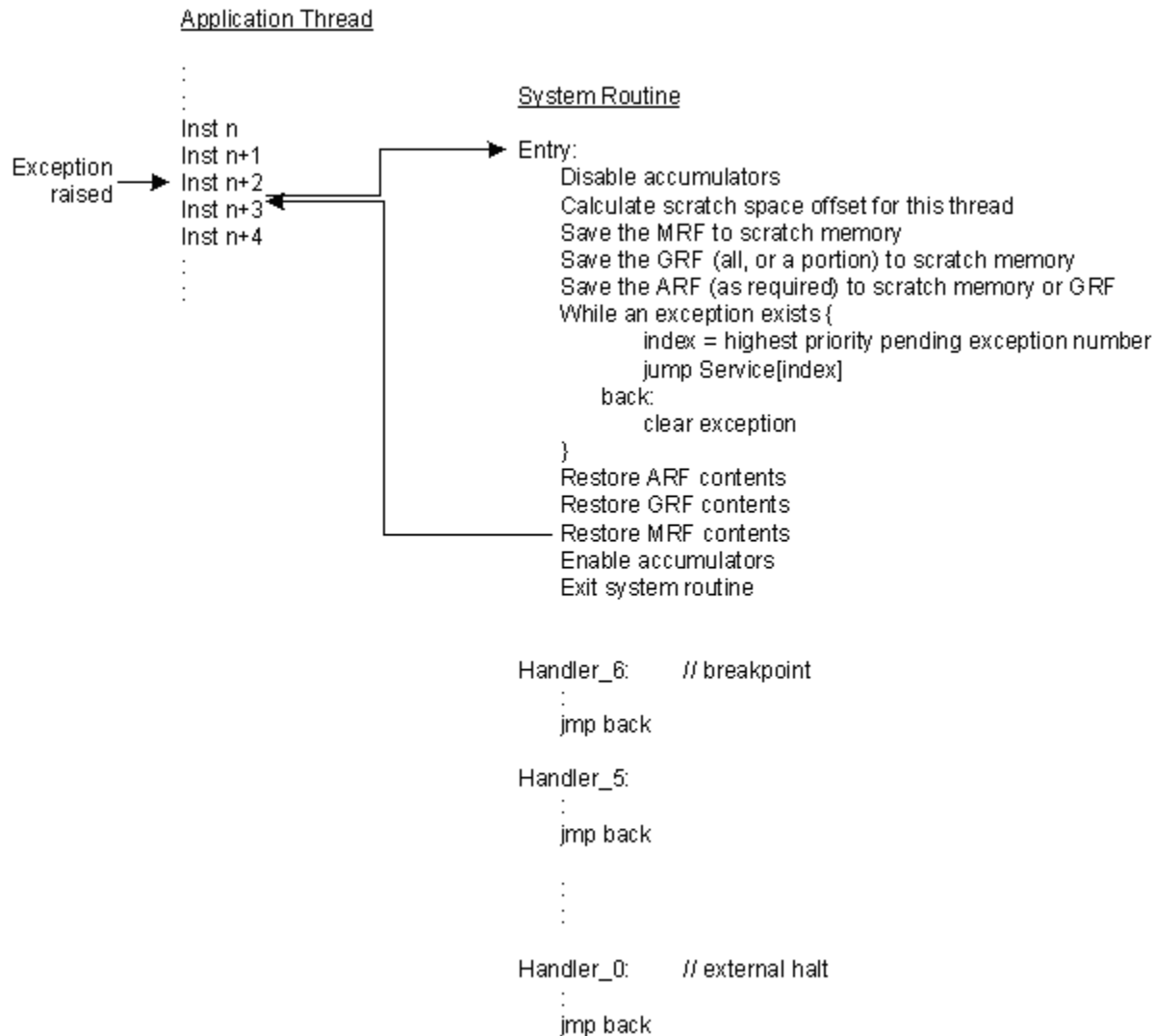
Exception-Related Architecture Registers

Exception-related registers are architecture registers cr0.0 through cr0.2. These registers are instantiated on a per-thread basis providing each hardware thread with unique control over exception recognition and handling. The registers provide the capability to mask exception types, determine the type of raised exception, store the return address, and control exiting from the System Routine back to the application thread.

Many of the bits in these registers are manipulated by both hardware and software. In all cases, the read/write operations by hardware and software occur at exclusive times in a thread's lifetime, thus there is no need for atomic read-modify-write operations when accessing these registers.

System Routine

The following diagram illustrates the basic flow of exception handling and the structure of the System Routine.



Invoking the System Routine

The System Routine is invoked in response to a raised exception. Once an exception is raised, no further instructions from the application thread are issued until the System Routine has executed and returned control back to the application thread.

After an exception is recognized by hardware, the EU saves the thread's IP into its AIP register (cr0.2), and then moves the System Routine offset, SIP, into the thread's IP register. At this point the next instruction issued for that thread is the first instruction of the System Routine.

The System Routine maintains the same execution priority, GRF register space, and thread state as the application thread from which it is invoked.

Due to assuming the same priority, there may be significant absolute time between an exception being raised and invoking the System Routine, as other higher priority threads within the EU continue to execute. From a thread's perspective, once an exception is recognized, the next instruction issued is from the System Routine.

At the time of System Routine invocation, there may still be outstanding registers in-flight from the application thread. Depending on the instruction sequence in the System Routine, an in-flight register may be referenced by the System Routine and cause a register-in-flight dependency. These dependencies are honored by the System Routine and may cause the System Routine to be suspended until the register retires.

Exception processing is not nested within the System Routine. If a future exception is detected while executing the System Routine, the exception is latched into cr0.1, but does not cause a nested re-invocation of the System Routine. The exception recognition hardware recognizes only one outstanding exception of each type; i.e., once a specific exception type is detected and latched in cr0.1, and until the exception is cleared, any further exception of that type is lost.

Accumulators are not natively preserved across the System Routine. To make sure the accumulators are in the identical state once control is returned to the application thread, the System Routine must either set the Accumulator Disable bit of cr0.0 before using any instruction that modifies an accumulator, or save and restore the accumulators (using GRF registers or system thread scratch memory) around the System Routine. Saving and restoring accumulators, including their extended precision bits, can be accomplished by a short series of moves and shifts of the accumulator register. Also note that the state of the Accumulator Disable bit itself must be preserved unless, by convention, the driver software limits its manipulation to only the System Routine.

Further, upon System Routine entry, the execution-related masks (Continue, Loop, If, and Active masks, contained in the Mask Register) will remain set as they were in the application thread. Thus only a subset of channels may be active for execution. To enable execution on all channels, the System Routine may choose to use the instruction option 'NoMask', or may choose to set the mask registers to the desired value so long as it saves/restores the original masks upon System Routine entry/exit.

Similarly there is no hardware mechanism to preserve flags, mask-stacks, or other architecture registers across the System Routine. The System Routine must ensure that these values are preserved (see the *Conditional Instructions Within the System Routine* section for related information).

Returning to the Application Thread

Prior to returning control to the application thread, the System Routine should clear the proper Exception Status and Control bit in cr0.1. Failure to do so forces the thread's execution to reenter the System Routine before any further instructions are executed from the application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset before exit.)

The System Routine may choose to loop within a single invocation of the System Routine until all pending exceptions are serviced, or may choose to service exceptions one at a time (a simpler solution, but less efficient).

The System Routine is exited, and control returned to the application thread, via a write to the 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.

The SIP used for GPGPU Context Save routine must follow these rules:

- a. All FC and MSG registers must write to odd GRF register and all other registers must write to even GRF register. This may change the context image and affects the context restore sequence accordingly.
- b. IEEE macro operations must never be used if preemption is desired.

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. 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.

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 provide 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.

CALLA is not recognized as a legal opcode. When calla is used, illegal opcode exception must be disabled.

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 . 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 0xFFFFFFFFD
#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'd 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:
jmp (1) r1.1:uw{NoMask}
mov (1) IP CRService_0 {NoMask}
mov (1) IP CRService_1 {NoMask}
mov (1) IP CRService_2 {NoMask}
...
mov (1) IP CRService_15{NoMask}
mov (1) IP Next

Service_0:

```

```
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).
...
Service_15:
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).

Exit:
// Restore f0.0.
// Restore other ARF registers (as required).
// Restore r12..r15.
// Restore r8..r11.
// Restore r4..r7.
// Restore r0..r3.
// Restore m8..m15.
// Restore m0..m7.
// Clear 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.

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		AddrMode = Indirect		SEND		MATH	Branch (2offsets)	Branch (1offset)	Imm Src[31:0]	Imm Src[63:0]
					AccessMode = Align16	AccessMode = Align1	AccessMode = Align16	AccessMode = Align1	MsgDesc Imm	MsgDesc Reg					
1	127	127	1	1					EOT						
	126	121	6	6											
	120	117	4	4	Src1.VertStride										
	118	118	1	1											
	115	114	2	2											
	113	112	2	2											
	111	111	1	1	Src1.ChanSel[7:4]										
	110	109	2	2	Src1.AddrMode										
	108	105	4	4	Src1.SrcMod										
	104	101	4	4	Src1.RegNum [7:0]				Src1.AddrSubRegNum						
3	100	100	1	1	Src1.SubRegNum [4]				Src1.AddrImm [8:4]						
	99	98	4	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]	
	95	95	1	1											
	94	91	4	4	Src1.SrcType										
	90	89	2	2	Src1.RegFile										
	88	85	4	4	Src0.VertStride										
	84	84	1	1											
	83	82	2	2											
	81	80	2	2	Src0.ChanSel[7:4] Src0.HorzStride Src0.ChanSel[7:4] Src0.HorzStride										
	79	79	1	1	Src0.AddrMode										
2	78	77	2	2	Src0.SrcMod										
	76	73	4	4	Src0.RegNum [7:0]				Src0.AddrSubRegNum						
	72	69	4	4	Src0.SubRegNum [4]				Src0.AddrImm [8:4]						
	68	68	1	1	Src0.ChanSel[3:0] Src0.SubRegNum [4:0] Src0.ChanSel[3:0] Src0.AddrImm [8:0]										
	67	64	4	4											
	63	63	1	1	Dst.AddrMode										
	62	61	2	2	Dst.HorzStride										
	60	57	4	4	Dst.RegNum [7:0]				Dst.AddrSubRegNum						
	56	53	4	4	Dst.SubRegNum [4]				Dst.AddrImm [8:4]						
	52	52	1	1	Dst.ChanEn[3:0] Dst.SubRegNum [4:0] Dst.ChanEn[3:0] Dst.AddrImm [8:0]										
0	51	48	4	4											
	47	47	1	1	Reserved										
	46	43	4	4	Src0.SrcType										
	42	41	2	2	Src0.RegFile										
	40	37	4	4	Dst.Type										
	36	35	2	2	Dst.RegFile										
	34	34	1	1	WECtrl										
	33	32	2	2	FlagRegNum/FlagSubRegNum				Same	Same	Same	Same	Same	Same	Same
	31	31	1	1	Saturate										
	30	30	1	1	DebugCtrl										
0	29	29	1	1	CmpCtrl										
	28	28	1	1	AccWrCtrl / BranchCtrl				Same	Same	Same	Same	Same	Same	Same
	27	24	4	4	CondModifier				SFID[3:0]	FC[3:0]	MBZ	MBZ			
	23	21	3	3	ExecSize										
	20	20	1	1	PredInv										
	19	18	4	4	PredCtrl										
	15	14	2	2	ThreadCtrl										
	13	12	2	2	QtrCtrl										
	11	11	1	1	NibCtrl										
	10	9	2	2	DepCtrl										
0	8	8	1	1	AccessMode				Same	Same	Same	Same	Same	Same	Same
	7	7	1	1	(reserved for Opcode)										
	7	6	0	7	Opcode				Same	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
2,3	1	127	127	1	<i>reserved</i>
	1	126	126	1	<i>reserved</i>
	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	<i>reserved</i>
	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	<i>reserved</i>
	8	83	76	8	Src0 Regnum
	3	75	73	3	Src0 Subregnum
	8	72	65	8	Src0 Swizzle
	1	64	64	1	Src0 RepCtrl
1	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	<i>reserved</i>
	1	35	35	1	<i>reserved</i>
	1	34	34	1	WECtrl
	2	33	32	2	FlagRegNum/FlagSubRegNum
0	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	<i>(reserved for Opcode)</i>
	7	6	0	7	Opcode

Native Instruction Layouts

This section describes the Execution Unit instruction formats for BDW.

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 BDW

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]			
113:112				Src1.HorzStride	
111	Src1.AddrMode				
110:109	Src1.SrcMod				
108:105	Src1.SrcMod	Src1.RegNum		Src1.AddrSubRegNum	
104:101					
100					
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]			

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
81:80				Src0.HorzStride	
79	Src0.AddrMode				
78:77	Src0.SrcMod				
76:73	Varies based on AddrMode and AccessMode	Src0.RegNum		Src0.AddrSubRegNum	
72:69					
68					
67:64		Src0.ChanSel[3:0]	Src0.ChanSel[3:0]		
63	Dst.AddrMode				
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					
52					
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				
27:24	CondModifier				
23:21	ExecSize				
20	PredInv				
19:16	PredCtrl				
15:14	ThreadCtrl				
13:12	QtrCtrl				

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
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

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description		
		send / sendc	math	Branch Instructions
127	Reserved	EOT		JIP[31:0]
126:125	Reserved	Imm[28:0] / 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			UIP[31:0] (2-offset branches)
94:91	Src1.SrcType			
90:89	Src1.RegFile			
88:85	Src0.VertStride			
84:80	Varies based on AccessMode			
79	Src0.AddrMode			
78:77	Src0.SrcMod			
76:64	Varies based on AddrMode and AccessMode			
63	Dst.AddrMode			
62:61	Varies based on AccessMode			
60:48	Varies based on AddrMode and AccessMode			
47	Reserved			
46:43	Src0.SrcType			
42:41	Src0.RegFile			
40:37	Dst.DstType			
36:35	Dst.RegFile			
34	MaskCtrl			

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description		
		send / sendc	math	Branch Instructions
33:32	FlagRegNum / FlagSubRegNum			
31	Saturate			
30				
29	CmptCtrl			
28	AccWrCtrl			
27:24	CondModifier	SFID[3:0]	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			

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 accommodate the new type encodings, providing a new source modifier interpretation for logical operations, widening the AddrSubRegNum fields to four bits to accommodate 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	<p>Access Mode. This field controls operand access for the instruction. It applies to all source and destination operands.</p> <p>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.</p> <p>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.</p> <p>0 = Align1 1 = Align16</p>
AccWrCtrl/ BranchCtrl	<p>Accumulator Write Control. Enable or disable implicitly writing results to the accumulator as well as to the destination.</p> <p>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.</p> <p>This bit should not be set if the accumulator is the explicit destination operand. Some instructions do not allow this option, including <i>mov</i> and <i>send</i>.</p> <p>Branch Control. Used by the <i>goto</i> instruction to control branching. See the <i>goto</i> instruction description for more information about BranchCtrl.</p>

Field	Description
CmptCtrl	<p>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.</p> <p>0 = No Compaction. No compaction. 128-bit native instruction supporting all instruction options.</p> <p>1 = CmptCtrl. Compaction is enabled. 64-bit compact instruction supporting only some instruction variations.</p>
CondModifier	<p>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.</p> <p>This field applies to all instructions except send, sendc, and math.</p> <p>0000b = Do not modify the flag register (normal)</p> <p>0001b = Zero or Equal (.z or .e)</p> <p>0010b = Not Zero or Not Equal (.nz or .ne)</p> <p>0011b = Greater than (.g)</p> <p>0100b = Greater than or equal (.ge)</p> <p>0101b = Less than (.l)</p> <p>0110b = Less than or equal (.le)</p> <p>0111b = Reserved</p> <p>1000b = Overflow (.o)</p> <p>1001b = Unordered with Computed NaN (.u)</p> <p>1010b-1111b = Reserved</p>
DepCtrl	<p>Destination Dependency Control. This field selectively disables destination dependency check and clear for this instruction.</p> <p>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.</p> <p>When NoDDClr is set, the destination register scoreboard is <i>not</i> 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.</p> <p>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.</p> <p>00b = Destination dependency checked and cleared (normal).</p> <p>01b = NoDDClr. Destination dependency checked but not cleared.</p> <p>10b = NoDDChk. Destination dependency not checked but cleared.</p> <p>11b = NoDDClr, NoDDChk. Destination dependency not checked and not cleared.</p> <p>See the Destination Hazard and the Use of NoDDClr sections for more information.</p>

Field	Description
EOT	End of Thread. For a <i>send</i> or <i>sendc</i> instruction, this bit controls thread termination. It is not used for other instructions. For a <i>send</i> or <i>sendc</i> 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 <i>send</i> and <i>sendc</i> 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. $ExecSize \times \text{largest data type size in bytes} \leq 64$. 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 <i>Width</i> must be $\leq ExecSize$.
FC	Function Control. Specifies the extended math function carried out by the <i>math</i> instruction. Not used for any other instruction. This field is in the same position as the CondModifier instruction field, so the <i>math</i> 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 <i>brc</i> (Branch Converging) instruction, both JIP and UIP can be contained in a register.
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.

Field	Description
NibCtrl	<p>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.</p> <p>0 = Use an odd 1/8th for DMask/VMask and ARF (first, third, fifth, or seventh depending on QtrCtrl).</p> <p>1 = Use an even 1/8th for DMask/VMask and ARF (second, fourth, sixth, or eighth depending on QtrCtrl).</p> <p>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.</p>
OpCode	<p>Operation Code. Contains the instruction operation code. Each opcode used is given a unique mnemonic. For example, the opcode 0x01 has the mnemonic <i>mov</i> indicating a Move instruction.</p> <p>See the Opcode Encoding section for details of opcode encoding.</p>

Field	Description
PredCtrl	<p>Predicate Control. This field, together with <i>PredInv</i>, 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.</p> <p>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.</p> <p>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.</p> <p>Predicate Control in Align16 access mode:</p> <p>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.</p> <p>Predicate Control in Align1 access mode:</p> <p>0000b = No predication (normal). 0001b = Predication with sequential flag channel mapping. 0010b = Predication with .anyv (f0.0 OR f0.1 on the same channel). 0011b = Predication with .allv (f0.0 AND f0.1 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). 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.</p>
PredInv	<p>Predicate Inverse. Together with <i>PredCtrl</i>, controls generation of the predication mask (PMask) for the instruction. When it is set and <i>PredCtrl</i> is not 0000b, predication uses the inverse of the predication bits produced by <i>PredCtrl</i>. In other words, the effect of <i>PredInv</i> happens after the effects of <i>PredCtrl</i>.</p> <p>This field is ignored if <i>PredCtrl</i> is 0000b; there is no predication.</p> <p>0 = +. Positive polarity of predication. Use the predication mask produced by <i>PredCtrl</i>. 1 = -. Negative polarity of predication. If <i>PredCtrl</i> is nonzero, invert the predication mask. PMask is the final predication mask produced by the effects of both fields.</p>

Field	Description				
QtrCtrl	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.				
	QtrCtrl	NibCtrl	ExecSize	Syntax	Description
	00b	0	8	1Q	Use first quarter for DMask/VMask. Use first half for everything else.
	01b	0	8	2Q	Use second quarter for DMask/VMask. Use second half for everything else.
	10b	0	8	3Q	Use third quarter for DMask/VMask. Use first half for everything else.
	11b	0	8	4Q	Use fourth quarter for DMask/VMask. Use second half for everything else.
	00b	0	16	1H	Use first half for DMask/VMask. Use all channels for everything else.
	10b	0	16	2H	Use second half for DMask/VMask. Use all channels for everything else.
	00b	0	4	1N	Use first 1/8th for DMask/VMask and ARF.
	00b	1	4	2N	Use second 1/8th for DMask/VMask and ARF.
	01b	0	4	3N	Use third 1/8th for DMask/VMask and ARF.
	01b	1	4	4N	Use fourth 1/8th for DMask/VMask and ARF.
	10b	0	4	5N	Use fifth 1/8th for DMask/VMask and ARF.
	10b	1	4	6N	Use sixth 1/8th for DMask/VMask and ARF.
	11b	0	4	7N	Use seventh 1/8th for DMask/VMask and ARF.
	11b	1	4	8N	Use eighth 1/8th for DMask/VMask and ARF.
	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 <i>send</i> or <i>sendc</i> instruction refers to the option of providing the message descriptor field DWord, of which bits 28:0 are used, in the first two words of the Address Register rather than as an immediate operand.				

Field	Description
Saturate	<p>Saturate. Enables or disables destination saturation.</p> <p>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.</p> <p>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.</p> <p>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].</p> <p>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.</p> <p>0 = No destination modification (normal). 1 = sat. Saturate the output.</p>
SFID	<p>Shared Function ID. Specifies a shared function that is the target of a <i>send</i> or <i>sendc</i> instruction. This field is not used for any other instructions. This field is in the same position as the CondModifier instruction field, so the <i>send</i> and <i>sendc</i> instructions do not support conditional modifiers.</p>
ThreadCtrl	<p>Thread Control. This field provides explicit control for thread switching.</p> <p>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.</p> <p>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.</p> <p>If this field is Atomic, the next instruction gets highest priority in the thread arbitration for the execution pipelines.</p> <p>Switch is designed primarily as a safety feature in case there are race conditions for certain instructions.</p> <p>00b = Normal Thread Control. Execution may or may not be preempted by another thread following this instruction.</p> <p>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).</p> <p>Atomic can be used with <i>send</i>, <i>sends</i>, <i>sendc</i>, <i>sendsc</i> ONLY.</p> <p>10b = Switch. Force a switch to another thread after this instruction and before the next instruction.</p> <p>11b = Reserved</p>

Field	Description
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 <i>brc</i> (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	<p>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.</p> <p>Allowed for any GRF register operand, destination or source. ARF registers cannot be accessed with indirect addressing.</p> <p>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.</p> <p>In the Align16 Access Mode, the low four bits of AddrImm are zero and do not appear in the instruction format.</p> <p>Format = S9</p> <p>Range = [-512, 511]</p>
AddrMode	<p>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.</p> <p>0 = Direct Addressing ("Direct"). Direct register addressing.</p> <p>1 = Indirect Addressing ("Register-Indirect" or "Indirect"). Register-indirect register addressing.</p> <p>Instructions with 3 source operands use Direct Addressing.</p>
AddrSubRegNum	<p>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.</p> <p>Format = U4</p> <p>Range = [0, 15]</p> <p>Some variations of Indirect Addressing use multiple address subregisters, where AddrSubRegNum determines the first subregister used.</p>

Field	Description
ChanEn	<p>Channel Enable. Dst.ChanEn, used only for the destination operand and only in the Align16 Access Mode.</p> <p>Provides four channel enable bits applied modulo four to all <i>ExecSize</i> channels. For example, 0xF enables all channels, 0 disables all channels, 0xA enables odd-numbered channels, and so on.</p> <p>The assembler mnemonics are x, y, z, and w for channels 0, 1, 2, and 3 respectively.</p> <p>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.</p> <p>For i in 0, 1, 2, 3:</p> <p>Bit i = 0 For channel j where $j \% 4 == i$, disable writing that channel.</p> <p>Bit i = 1 For channel j where $j \% 4 == i$, enable writing that channel.</p>
ChanSel	<p>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.</p> <p>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.</p> <p>The 2-bit Channel Selection field for each channel within the group of 4 is defined as:</p> <p>00b = x. Channel 0 is selected for the corresponding execution channel.</p> <p>01b = y. Channel 1 is selected for the corresponding execution channel.</p> <p>10b = z. Channel 2 is selected for the corresponding execution channel.</p> <p>11b = w. Channel 3 is selected for the corresponding execution channel.</p> <p>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.</p>
Desc	<p>Message descriptor. A 31 bit immediate message descriptor for <i>send</i>, <i>sendc</i>, <i>sends</i> and <i>sendsc</i> instructions. This field is not used for other instructions.</p>

Field	Description
DstType	<p>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 <i>send</i> or <i>sendc</i> instruction, this field applies to CurrDst, the current destination operand. Three source instructions use a 3-bit encoding that allows fewer data types.</p> <p>Encoding for one or two source instructions:</p> <p>0000b = :ud. Unsigned Doubleword integer. 0001b = :d. Signed Doubleword integer. 0010b = :uw. Unsigned Word integer. 0011b = :w. Signed Word integer. 0100b = :ub. Unsigned Byte integer. 0101b = :b. Signed Byte integer. 0110b = :df. Double precision Float (64-bit). 0111b = :f. Single precision Float (32-bit). 1000b = :uq. Unsigned Quadword integer. 1001b = :q. Signed Quadword integer. 1010b = :hf. Half Float (16-bit). 1011b to 1111b = Reserved.</p> <p>Encoding for three source instructions:</p> <p>000b = :f. Single precision Float (32-bit). 001b = :d. Signed Doubleword integer. 010b = :ud. Unsigned Doubleword integer. 011b = :df. Double precision Float (64-bit). 100b = :hf. Half precision Float (16-bit). 101b - 111b. Reserved.</p>
ExDesc	<p>Extended message descriptor. A 32 bit immediate extended message descriptor for <i>send</i>, <i>sendc</i>, <i>sends</i> and <i>sendsc</i> instructions. This field is not used for other instructions.</p>
HorzStride	<p>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.</p> <p>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.</p> <p>A horizontal stride of 1 indicates that elements are adjacent within a row.</p> <p>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.</p> <p>This field applies to both source and destination register operands.</p> <p>This field is used with both direct and indirect addressing.</p> <p>00b = 0 Elements 01b = 1 Element 10b = 2 Elements 11b = 4 Elements</p> <p>See the Register Region Restrictions section for rules that constrain <i>HorzStride</i> in relation to other region parameters.</p>
Imm[28:0]	<p>A 29-bit immediate message descriptor for a <i>send</i> or <i>sendc</i> instruction. This field is not used for other instructions.</p>

Field	Description
Imm32	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>See the Numeric Data Types section for information about data types and their ranges.</p>
Imm64	<p>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).</p>
RegFile	<p>Register File. Select a source or destination register file or indicate an immediate source operand:</p> <p>00b = ARF Architecture Register File. Only allowed for src0 or destination.</p> <p>01b = GRF General Register File. Allowed for any source or destination.</p> <p>10b = Reserved.</p> <p>11b = Immediate operand. Only allowed for the last source operand. Not allowed for the 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.</p>
RegNum	<p>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.</p> <p>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.</p> <p>This field is present for the direct addressing mode and not present for indirect addressing. This field applies to both source and destination operands.</p> <p>Format = U8, if RegFile = GRF.</p> <p>0x00 to 0x7F = Register number in the range [0, 127].</p> <p>0x80 to 0xFF = Reserved.</p> <p>Format = 8-bit encoding, if RegFile = ARF.</p> <p>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.</p>

Field	Description
RepCtrl	<p>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.</p> <p>0 = No replication. 1 = Replicate across all channels.</p>
SelReg32Desc	<p>Select Reg32 for message descriptor. In <i>sends</i> or <i>sendsc</i> instruction, refers to the selection of Reg32 for the message descriptor field.</p> <p>0 = Desc. 1 = Reg32. First Dword of Address Register is used for message descriptor.</p>
SrcMod	<p>Source Modifier. Specify any numeric (normally) or logical (for logic instructions) modification to a source value before delivery to the execution pipe.</p> <p>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).</p> <p>When used with logic instructions (<i>and</i>, <i>not</i>, <i>or</i>, <i>xor</i>), this field indicates whether the source bits are inverted (bitwise NOT) before delivery to the execution pipe, regardless of the source type.</p> <p>This field only applies to source operands. It does not apply to the destination.</p> <p>This field is not present for an immediate source operand.</p> <p>For no modification, there is no assembler notation or syntax.</p> <p>Encoding for all instructions other than logic instructions:</p> <p>00b = No modification (normal). 01b = (abs). Absolute value. 10b = -. Negation. 11b = -(abs). Negation of the absolute value (forced negative value).</p> <p>Encoding for logic instructions:</p> <p>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.</p>
SrcType	<p>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.</p> <p>The bits of a source operand are interpreted as the identified numeric data type, rather than coerced into a type implied by the operator.</p> <p>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.</p> <p>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.</p>

Field	Description
	<p>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 Register Type Encoding and the Source Immediate Type Encoding.</p> <p>The Source Register Type Encoding and Source Immediate Type Encoding lists apply to instructions with one or two source operands.</p> <p>Source Register Type Encoding:</p> <p>0000b = UD. Unsigned Doubleword integer. 0001b = D. Signed Doubleword integer. 0010b = UW. Unsigned Word integer. 0011b = W. Signed Word integer. 0100b = UB. Unsigned Byte integer. 0101b = B. Signed Byte integer. 0110b = DF. Double precision Float (64-bit). 0111b = F. Single precision Float (32-bit). 1000b = UQ. Unsigned Quadword integer. 1001b = Q. Signed Quadword integer. 1010b = HF. Half Float (16-bit). 1011b to 1111b = Reserved.</p> <p>Source Immediate Type Encoding:</p> <p>0000b = UD. Unsigned Doubleword integer. 0001b = D. Signed Doubleword integer. 0010b = UW. Unsigned Word integer. 0011b = W. Signed Word integer. 0100b = UV. Packed Unsigned Half-Byte Integer Vector, 8 x 4-Bit Unsigned Integer. 0101b = VF. Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number. 0110b = V. Packed Signed Half-Byte Integer Vector, 8 x 4-Bit Signed Integer. 0111b = F. Single precision Float (32-bit). 1000b = UQ. Unsigned Quadword integer. 1001b = Q. Signed Quadword integer. 1010b = DF. Double precision Float (64-bit). 1011b = HF. Half Float (16-bit). 1100b to 1111b = Reserved.</p> <p>Three source instructions use one SrcType field for all source operands, with a 3-bit encoding that allows fewer data types:</p> <p>Encoding for three source instructions:</p> <p>000b = :f. Single precision Float (32-bit). 001b = :d. Signed Doubleword integer. 010b = :ud. Unsigned Doubleword integer. 011b = :df. Double precision Float (64-bit). 100b = :hf. Half precision Float (16-bit). 101b - 111b. Reserved.</p>

Field	Description
SubRegNum	<p>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.</p> <p>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.</p> <p>This field is present for the direct addressing mode and not present for indirect addressing. This field applies to both source and destination operands.</p> <p>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].</p> <p>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.</p> <p>Format = 5-bit encoding, if RegFile = ARF.</p> <p>See the ARF Registers Overview section and the sections for individual ARF registers for details.</p>

Field	Description
VertStride	<p>Vertical Stride. The vertical stride of a source operand's register region in units of data element size.</p> <p>Supported values are 0, powers of 2 from 1 to 32, and a special encoding used for indirect addressing in Align1 mode.</p> <p>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).</p> <p>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.</p> <p>This field only applies to source operands. It does not apply to the destination.</p> <p>For Align16 access mode, only encodings of 0000b, 0010b, and 0011b are allowed. Other codes are reserved.</p> <p>Note 1: A Vertical Stride larger than 32 is not allowed due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total).</p> <p>Note 2: In Align16 access mode, as encoding 1111b is reserved, only single-index indirect addressing is supported.</p> <p>Note 3: If indirect addressing is supported for src1, the encoding 1111b is reserved for src1 and only single-index indirect addressing is supported.</p> <p>Note 4: The encoding 0010b is used for 64-bit operands (types DF, Q, or UQ).</p> <p>0000b = 0 Elements</p> <p>0001b = 1 Element. Align1 mode only.</p> <p>0010b = 2 Elements</p> <p>0011b = 4 Elements</p> <p>0100b = 8 Elements. Align1 mode only.</p> <p>0101b = 16 Elements (applies to byte or word operands only). Align1 mode only.</p> <p>0110b = 32 Elements (applies to byte operands only). Align1 mode only.</p> <p>0111b-1110b = Reserved.</p> <p>1111b = VxH or Vx1 mode (only valid for register-indirect addressing in Align1 mode).</p>
Width	<p>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.</p> <p>This field only applies to source operands. It does not apply to the destination.</p> <p>000b = 1 Element</p> <p>001b = 2 Elements</p> <p>010b = 4 Elements</p> <p>011b = 8 Elements</p> <p>100b = 16 Elements</p> <p>101b-111b = Reserved</p> <p>Note that with ExecSize of 32, because the maximum Width is 16, there are at least two rows in a source region.</p>

EU Compact Instructions

On receiving an instruction with bit 29 (CmptCtrl) set, HW recognizes it as a 64-bit compact instruction. Hardware then uses the index fields inside the compact instruction to lookup values in the associated compaction tables, then uses the table outputs along with other fields in the compact instruction to reconstruct the 128-bit native-sized instruction.

All flow control instructions use the new offset format, a signed 32-bit offset in units of bytes.

The native 128-bit instruction format provides access to all instruction options. Only some instruction options and combinations of instruction options can be represented in the compact instruction formats.

Which native instructions can be represented as compact instructions and the details of the compact instruction formats and the compaction tables used may change with each processor generation.

In the following instruction format tables the Mapping Bits and Mapping Description columns describe the mappings into native instruction fields.

EU Compact Instruction Format

The following table describes the EU compact instruction format for BDW. BDW adds support for compacting instructions with three source operands.

BDW supports instruction compaction for 3-source instructions as well as for 1 or 2-source instructions. The next two tables provide the BDW compact instruction formats for 1 or 2-source instructions first and then for 3-source instructions.

The BDW compact instruction format for 1 or 2-source instructions is essentially identical to the compact instruction format for earlier generations, but the compact fields expand to somewhat different fields in the native instruction format, as the native instruction format changed for BDW.

GEN Compact Instruction Format BDW 1 or 2 Source Operands

Bits	Size	Mapping Bits	Compact Name	Mapping Description
63:56	8	108:101 (Not Imm.) or 103:96 (Imm.)	Src1.RegNum	Src1.RegNum if not immediate. Imm32[7:0] if immediate.
55:48	8	76:69	Src0.RegNum	Src0.RegNum.
47:40	8	60:53	Dst.RegNum	Dst.RegNum.
39:35	5	120:109 (Not Imm.) or 127:104 (Imm.)	Src1Index	If not an immediate operand, lookup one of 32 12-bit values that maps to bits 120:109. That value is used (from MSB to LSB) for the Src1.VertStride, various Src1 bit fields based on AccessMode (Src1.ChanSel[7:4], Src1.Width, Src1.HorzStride), Src1.AddrMode, and Src1.SrcMod bit fields. If an immediate operand, there is no lookup. Determines bits 127:104 (Imm32[31:8]) as follows: map bits 39:35 directly to bits 108:104. Sign extend to fill bits 127:109. Compact format bit 39 is thus copied to all of bits 127:108 for an immediate operand.
34:30	5	88:77	Src0Index	Lookup one of 32 12-bit values. That value is used (from MSB to LSB) for the Src0.VertStride, various Src0 bit fields based on AccessMode (Src0.ChanSel[7:4], Src0.Width, Src0.HorzStride), Src0.AddrMode, and Src0.SrcMod bit fields. Note that this field spans a DWord boundary within the QWord compacted instruction.
29	1	29	CmptCtrl	Compaction Control. The same in both the compact and native formats: 0: Regular instruction, not compacted. 1: Compacted instruction.
28	1	Not mapped.	Reserved	Not mapped.
27:24	4	27:24	CondModifier	CondModifier. The same in both the compact and native formats.
23	1	28	AccWrCtrl	AccWrCtrl.
22:18	5	100:96, 68:64, 52:48	SubRegIndex	Lookup one of 32 15-bit values. That value is used (from MSB to LSB) for various fields for Src1, Src0, and Dst, including ChanEn/ChanSel, SubRegNum, and AddrImm[4] or AddrImm[4:0], depending on AddrMode and AccessMode.
17:13	5	63:61, 94:89, 46:35	DataTypeIndex	Lookup one of 32 21-bit values. That value is used (from MSB to LSB) for the Dst.AddrMode, Dst.HorzStride, Src1.SrcType, Src1.RegFile, Src0.SrcType, Src0.RegFile, Dst.DstType, and Dst.RegFile bit fields.
12:8	5	33:32, 31, 23:12, 10:9, 34, 8	ControlIndex	Lookup one of 32 19-bit values. That value is used (from MSB to LSB) for the FlagRegNum/FlagSubRegNum, Saturate, ExecSize, PredInv, PredCtrl, ThreadCtrl, QtrCtrl, DepCtrl, MaskCtrl, and AccessMode bit fields.
6:0	7	6:0	Opcode	Opcode. The same in both the compact and native formats.

GEN Compact Instruction Format 3 Source Operands

Bits	Size	Mapping Bits	Compact Name	Mapping Description
63:57	7	124:118	Src2.RegNum[6:0]	Src2.RegNum[6:0]. The SourceIndex field in the compact instruction determines Src2.RegNum[7].
56:50	7	103:97	Src1.RegNum[6:0]	Src1.RegNum[6:0]. The SourceIndex field in the compact instruction determines Src1.RegNum[7].
49:43	7	82:76	Src0.RegNum[6:0]	Src0.RegNum[6:0]. The SourceIndex field in the compact instruction determines Src0.RegNum[7].
42:40	3	117:115	Src2.SubRegNum	Src2.SubRegNum.
39:37	3	96:94	Src1.SubRegNum	Src1.SubRegNum.
36:34	3	75:73	Src0.SubRegNum	Src0.SubRegNum.
33	1	106	Src2.RepCtrl	Src2.RepCtrl.
32	1	85	Src1.RepCtrl	Src1.RepCtrl.
31	1	31	Saturate	Saturate.
29	1	29	CmptCtrl	Compaction Control. The same in both the compact and native formats: 0: Regular instruction, not compacted. 1: Compacted instruction.
28	1	64	Src0.RepCtrl	Src0.RepCtrl.
27:19	9	Not mapped	Reserved	Not mapped.
18:12	7	63:56	Dst.RegNum[6:0]	Dst.RegNum[7:0] with MSB of zero and [6:0] from the compact instruction.
11:10	2	125, 104, 83, 114:107, 93:86, 72:65, 55:49, 48:43, 42:37	SourceIndex	Lookup one of four 46-bit values. That value is used (from MSB to LSB) for the Src2.RegNum[7], Src1.RegNum[7], Src0.RegNum[7], Src2.ChanSel, Src1.ChanSel, Src0.ChanSel, Dst.SubRegNum, Dst.ChanEnable, Dst.DstType, SrcType, Src2.Modifier, Src1.Modifier, and Src0.Modifier bit fields.
9:8	2	34, 33:32, 28:8	ControlIndex	Lookup one of four 24-bit values. That value is used (from MSB to LSB) for the MaskCtrl, FlagRegNum/FlagSubRegNum, AccWrCtrl, CondModifier, ExecSize, PredInv, PredCtrl, ThreadCtrl, QtrCtrl, NibCtrl, DepCtrl, and AccessMode bit fields.
7	1	Not mapped	Reserved	Not mapped.
6:0	7	6:0	Opcode	Opcode. The same in both the compact and native formats.

The following diagrams are an alternate presentation of the BDW compact instruction formats.

GEN Compact Instruction Format 1 or 2 Source Operands

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	Description	Bits in 128bits Format	Description (Imm. Src0 or Src1)	Bits in 128bits Format (Imm. Src0 or Src1)
1	8	63	56	8	Src1 RegNum	[108:101]	Imm[23:16] Imm[7:0]	[119:112] [103:96]
	8	55	48	8	Src0 RegNum	[76:69]	Src0 RegNum	[76:69]
	8	47	40	8	Dst RegNum	[60:53]	Dst RegNum	[60:53]
	5	39	35	5	Src1Index[4:0]	[120:109]	Src1Index[4:0]	[127:120] [111:104]
	3	34	32	3				
0	2	31	30	2	Src0Index[4:0]	[88:77]	Src0Index[4:0]	[88:77]
	1	29	29	1	CmpCtrl	[29]	CmpCtrl	[29]
	1	28	28	1	Reserved		Reserved	
	4	27	24	4	CondModifier	[27:24]	CondModifier	[27:24]
	1	23	23	1	AccWrCtrl	[28]	AccWrCtrl	[28]
	5	22	18	5	SubRegIndex[4:0]	[100:96] [68:64] [52:48]	SubRegIndex[4:0]	[100:96] [68:64] [52:48]
	5	17	13	5	DataTypeIndex[4:0]	[63:61] [94:89] [46:35]	DataTypeIndex[4:0]	[63:61] [94:89] [46:35]
	5	12	8	5	ControllIndex[4:0]	[33:32] [31] [23:12] [10:9] [34] [8]	ControllIndex[4:0]	[33:32] [31] [23:12] [10:9] [34] [8]
	1	7	7	1	DebugCtrl	[30]	DebugCtrl	[30]
	7	6	0	7	Opcode	[6:0]	Opcode	[6:0]

GEN Compact Instruction Format 3 Source Operands

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	Description	Bits in 128bits Format
1	7	63	57	7	Src2 RegNum	[124:118]
	7	56	50	7	Src1 RegNum	[103:97]
	7	49	43	7	Src0 RegNum	[82:76]
	3	42	40	3	Src2 SubRegNum	[117:115]
	3	39	37	3	Src1 SubRegNum	[96:94]
	3	36	34	3	Src0 SubRegNum	[75:73]
	1	33	33	1	Src2 Replicate	[106]
	1	32	32	1	Src1 Replicate	[85]
0	1	31	31	1	Saturate	[31]
	1	30	30	1	DebugCtrl	[30]
	1	29	29	1	CmptCtrl	[29]
	1	28	28	1	Src0 Replicate	[64]
	9	27	19	9	Reserved	
	7	18	12	7	Dst RegNum	[63:56]
	2	11	10	2	SourceIndex[1:0]	[125][104][83][114:107][93:86][72:65][55:49][46:43][42:37]
	2	9	8	2	ControlIndex[1:0]	[34][33:32][28][27:24][23:21][20:14][13:12][11][10:8]
	1	7	7	1	Reserved	
	7	6	0	7	Opcode	[6:0]

EU Instruction Compaction Tables

The following four tables describe the mappings for the ControlIndex, DataTypeIndex, SubRegIndex, Src0Index, and Src1Index fields in the 1 or 2 source operand compact instruction format for BDW.

ControlIndex Compact Instruction Field Mappings 1 or 2 Source Operands

ControlIndex	19-Bit Mapping	Mapped Meaning
0	0000000000000000010	Align1 We (1) f0.0
1	0000100000000000000	Align1 (4) f0.0
2	0000100000000000001	Align16 (4) f0.0
3	0000100000000000010	Align1 We (4) f0.0
4	0000100000000000011	Align16 We (4) f0.0
5	0000100000000000100	Align1 NoDDClr (4) f0.0
6	0000100000000000101	Align16 NoDDClr (4) f0.0
7	0000100000000000111	Align16 We NoDDClr (4) f0.0
8	0000100000000001000	Align1 NoDDChk (4) f0.0
9	0000100000000001001	Align16 NoDDChk (4) f0.0
10	0000100000000001101	Align16 NoDDClr, NoDDChk (4) f0.0
11	0000110000000000000	Align1 Q1 (8) f0.0
12	0000110000000000001	Align16 Q1 (8) f0.0
13	0000110000000000010	Align1 We Q1 (8) f0.0
14	0000110000000000011	Align16 We Q1 (8) f0.0
15	0000110000000000100	Align1 NoDDClr Q1 (8) f0.0
16	0000110000000000101	Align16 NoDDClr Q1 (8) f0.0
17	0000110000000000111	Align16 We NoDDClr Q1 (8) f0.0
18	0000110000000001001	Align16 NoDDChk Q1 (8) f0.0
19	0000110000000001101	Align16 NoDDClr, NoDDChk Q1 (8) f0.0
20	0000110000000010000	Align1 Q2 (8) f0.0
21	0000110000100000000	Align1 Q1 +f.xyzw (8) f0.0
22	0001000000000000000	Align1 H1 (16) f0.0
23	0001000000000000010	Align1 We H1 (16) f0.0
24	0001000000000000100	Align1 NoDDClr H1 (16) f0.0
25	0001000000100000000	Align1 H1 +f.xyzw (16) f0.0
26	0010110000000000000	Align1 Q1 (8) .sat f0.0
27	0010110000000001000	Align1 Q2 (8) .sat f0.0
28	0011000000000000000	Align1 H1 (16) .sat f0.0
29	0011000000100000000	Align1 H1 +f.xyzw (16) .sat f0.0
30	0101000000000000000	Align1 H1 (16) f0.1
31	0101000000100000000	Align1 H1 +f.xyzw (16) f0.1

Data Type Index Compact Instruction Field Mappings 1 or 2 Source Operands

Data Type Index	21-Bit Mapping	Mapped Meaning
0	001000000000000000000001	r:ud a:ud a:ud <1> dir
1	001000000000000010000000	a:ud r:ud a:ud <1> dir
2	001000000000000010000001	r:ud r:ud a:ud <1> dir
3	001000000000001100000001	r:ud i:ud a:ud <1> dir
4	00100000000001010111101	r:f r:d a:ud <1> dir
5	00100000000101110111101	r:f i:vf a:ud <1> dir
6	00100000000111010000001	r:ud r:f a:ud <1> dir
7	0010000000011101000101	r:d r:f a:ud <1> dir
8	00100000000111010111101	r:f r:f a:ud <1> dir
9	001000000100000010000001	r:ud r:ud r:ud <1> dir
10	001000001100000010000000	a:ud r:ud i:ud <1> dir
11	001000001100000010000001	r:ud r:ud i:ud <1> dir
12	0010000101000101000101	r:d r:d r:d <1> dir
13	0010000111000101000100	a:d r:d i:d <1> dir
14	0010000111000101000101	r:d r:d i:d <1> dir
15	00101110000111010111101	r:f r:f a:f <1> dir
16	001011101011100011101	r:f a:f r:f <1> dir
17	001011101011101011100	a:f r:f r:f <1> dir
18	001011101011101011101	r:f r:f r:f <1> dir
19	001011111011101011100	a:f r:f i:f <1> dir
20	00000000000100000001100	a:w a:ub a:ud <0> dir
21	00100000000000001011101	r:f r:ud a:ud <1> dir
22	0010000000000101000101	r:d r:d a:ud <1> dir
23	001000000100000010000000	a:ud r:ud r:ud <1> dir
24	0010000101000101000100	a:d r:d r:d <1> dir
25	0010000111000100000100	a:d a:d i:d <1> dir
26	0010010010010000001001	r:uw a:uw r:uw <1> dir
27	001010111011101011101	r:f r:f i:vf <1> dir
28	001011111011101011101	r:f r:f i:f <1> dir
29	001001111001101001100	a:w r:w i:w <1> dir
30	0010010010010010010000	a:uw r:uw r:uw <1> dir
31	0010010110010010010000	a:uw r:uw i:uw <1> dir

SubRegIndex Compact Instruction Field Mappings 1 or 2 Source Operands BDW

SubRegIndex	15-Bit Mapping	Mapped Meaning
0	0000000000000000	0 0 0
1	0000000000000001	0.x 0.xx 0.xx
2	000000000001000	8 0 0
3	000000000001111	0.xyzw 0.xx 0.xx
4	00000000010000	16 0 0
5	000000100000000	0 4 0
6	000001000000000	0 8 0
7	000001100000000	0 12 0
8	000010000000000	0 16 0
9	00001000010000	16 16 0
10	000010100000000	0 20 0
11	001000000000000	0 0 4
12	001000000000001	0.x 0.xx 0.xy
13	001000010000001	0.x 0.xy 0.xy
14	001000010000010	0.y 0.xy 0.xy
15	001000010000011	0.xy 0.xy 0.xy
16	001000010000100	0.z 0.xy 0.xy
17	001000010000111	0.xyz 0.xy 0.xy
18	001000010001000	0.w 0.xy 0.xy
19	001000010001110	0.yzw 0.xy 0.xy
20	001000010001111	0.xyzw 0.xy 0.xy
21	001000110000000	0 12 4
22	001000111101000	0.w 0.ww 0.xy
23	010000000000000	0 0 8
24	010000110000000	0 12 8
25	011000000000000	0 0 12
26	011110010000111	0.xyz 0.xy 0.ww
27	100000000000000	0 0 16
28	101000000000000	0 0 20
29	110000000000000	0 0 24
30	111000000000000	0 0 28
31	11100000011100	28 0 28

Src0Index or Src1Index Compact Instruction Field Mappings 1 or 2 Source Operands BDW

Src0Index or Src1Index	12-Bit Mapping	Mapped Meaning
0	000000000000	dir <0;1,0>
1	000000000010	(-) dir <0;1,0>
2	000000010000	dir <0;>.zx
3	000000010010	(-) dir <0;>.zx
4	000000011000	dir <0;>.wx
5	000000100000	dir <0;>.xy
6	000000101000	dir <0;>.yy
7	000001001000	dir <0;4,1>
8	000001010000	dir <0;>.zz
9	000001110000	dir <0;>.zw
10	000001111000	dir <0;8,4> / dir <0;>.ww
11	001100000000	dir <4;>.xx
12	001100000010	(-) dir <4;>.xx
13	001100001000	dir <4;>.yx
14	001100010000	dir <4;>.zx
15	001100010010	(-) dir <4;>.zx
16	001100100000	dir <4;>.xy
17	001100101000	dir <4;>.yy
18	001100111000	dir <4;>.wy
19	001101000000	dir <4;4,0>
20	001101000010	(-) dir <4;4,0>
21	001101001000	dir <4;>.yz
22	001101010000	dir <4;>.zz
23	001101100000	dir <4;>.xw
24	001101101000	dir <4;>.yw
25	001101110000	dir <4;>.zw
26	001101110001	(abs) dir <4;>.zw
27	001101111000	dir <4;>.ww
28	010001101000	dir <8;8,1>
29	010001101001	(abs) dir <8;8,1>
30	010001101010	(-) dir <8;8,1>
31	010110001000	dir <16;16,1>

The following tables describe the mappings for the ControlIndex and SourceIndex fields in the 3 source operand compact instruction format for BDW.

ControlIndex Compact Instruction Field Mappings 3 Source Operands

ControlIndex	24-Bit Mapping	Mapped Meaning
0	100000000110000000000001	(8) Q1 NoMask Align16
1	000000000110000000000001	(8) Q1 Align16
2	000000001000000000000001	(16) H1 Align16
3	000000001000000000100001	(16) H2 Align16

SourceIndex Compact Instruction Field Mappings 3 Source Operands

SourceIndex	46-Bit Mapping	Mapped Meaning
0	0001110010011100100111001000001111000000000000	No Negation
1	0001110010011100100111001000001111000000000010	Negate Src0
2	0001110010011100100111001000001111000000001000	Negate Src1
3	0001110010011100100111001000001111000000100000	Negate Src2

Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

The opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxxb are reserved.

Note: Opcodes appear in the overall [Instruction Set Summary Table](#) as well. The following subsections still serve the purpose of describing various instruction groups.

Move and Logic Instructions

This instruction group has an opcode format of 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*, and *asr*) share the common 4 MSBs in the form of 0001xxxb. Bit 2 indicates arithmetic or logic shift (0 = logic, 1 = arithmetic). Bit 1 is always 0 (which is reserved for future extension to support rotation shift as 0 = shift, 1 = rotate). Bit 0 indicates the shift direction (0 = right, 1 = left).
- 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			
14	0x0E	Reserved			
15	0x0F	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			

Opcode		Instruction	Description	#src	#dst
dec	hex				
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	<i>Reserved</i>			

Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	jmp	Jump indexed	1	0
33	0x21	brd	Branch - Diverging	1	0
34	0x22	if	If	0/2	0
35	0x23	brc	Branch - Converging	1	-
36	0x24	else	Else	1	0
37	0x25	endif	End if	0	0
38	0x26	<i>Reserved</i>			
39	0x27	<i>Reserved</i>			
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	<i>Reserved</i>			
52	0x34	<i>Reserved</i>			
53-55	0x35-0x37	<i>Reserved</i>			
56	0x38	math	Math functions for extended math pipeline	1/2	1/2
57-63	0x39-0x3F	<i>Reserved</i>			

Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

Parallel Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
64	0x40	add	Component-wise addition	2	1
65	0x41	mul	Component-wise multiply	2	1
66	0x42	avg	Component-wise average of the two source operands	2	1
67	0x43	frc	Component-wise floating point truncate-to-minus-infinity fraction	1	1
68	0x44	rndu	Component-wise floating point rounding up (ceiling)	1	1
69	0x45	rndd	Component-wise floating point rounding down (floor)	1	1
70	0x46	rnde	Component-wise floating point rounding toward nearest even	1	1
71	0x47	rndz	Component-wise floating point rounding toward zero	1	1
72	0x48	mac	Component-wise multiply accumulate	2	1
73	0x49	mach	multiply accumulate high	2	1
74	0x4A	lzd	leading zero detection	1	1
75	0x4B	fbh	Find first 1 for UD from msb side, or first 1/0 for D.	1	1
76	0x4C	fbl	First first 1 for UD from lsb side	1	1
77	0x4D	cbit	Count bits set	1	1
78	0x4E	addc	Integer add with carry	2	1 + acc.
79	0x4F	subb	integer subtract with borrow	2	1 + acc.

Vector Arithmetic Instructions

This instruction group has an opcode format of 101xxxxb.

Vector Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
80	0x50	sad2	2-wide sum of absolute difference	2	1
81	0x51	sada2	2-wide sad accumulate	2	1
82-83	0x52-0x53	<i>Reserved</i>			
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	<i>Reserved</i>			
89	0x59	line	Component-wise line equation computation (a multiply-add)	2	1
90	0x5A	pln	Component-wise floating point plane equation computation (a multiply-multiply-add)	2	1
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	<i>fmam(madm)</i>	Component-wise floating point fused multiply and add for macro operations.	3	1
94-95	0x5E-0x5F	<i>Reserved</i>			

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	<i>Reserved</i>			
126	0x7E	nop	No-op	0	0
127	0x7F	<i>Reserved</i>			

Native Instruction BNF

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

Instruction Groups

<Instruction>::= <UnaryInstruction>

|<BinaryAccInstruction>

|<BinaryInstruction>

|<TriInstruction>

|<JumpInstruction>

|<BranchLoopInstruction>

|<ElseInstruction>

|<BreakInstruction>

|<MaskControlInstruction>

|<TriInstruction2>

|<CallInstruction>

|<BranchConvInstruction>

|<BranchDivInstruction>

|<MathInstruction>

|<SyncInstruction>

|<SpecialInstruction>

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

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

<UnaryOp>::= "mov" | "frc" | "rndu" | "rnda" | "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"

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

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>::=":df" | ":f" | ":ud" | ":d" | ":uw" | ":w" | ":ub" | ":b" | ":uq" | ":q" | ":hf"
 <SrcImmType>::= <SrcType> | ":v" | ":vf" | ":uv"
 <DstType>::= <SrcType>

Write Mask

```

<WriteMask>::= ""
| "." "x" | "." "y" | "." "z" | "." "w"
| "." "xy" | "." "xz" | "." "xw" | "." "yz" | "." "yw" | "." "zw"
| "." "xyz" | "." "xyw" | "." "xzw" | "." "yzw"
| "." "xyzw"

```

Swizzle Control

```

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

```

Immediate Values

```

<ImmAddrNum>::= "-512" ... "511"
<Imm64> ::= "0.0" ... "±1.0*2-1024...1023" | "0" ... "264-1" | "-263" ... "263-1"
<Imm32>::= "0.0" ... "±1.0*2-128...127" | "0" ... "232-1" | "-231" ... "231-1"
<Imm16>::= "0" ... "216-1" | "-215" ... "215-1"
<ImmDesc>::= "0" ... "232-1"

```

Predication and Modifiers

Instruction Predication

<Predicate>::=""

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

<PredState>::=""

| "+"

| "-"

<PredCntrl>::=""

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

| ".any2h" | ".all2h"

| ".any4h" | ".all4h"

| ".any8h" | ".all8h"

| ".any16h" | ".all16h"

| ".anyv" | ".allv"

| ".any32h" | ".all32h"

Source Modification

<SrcModifier>::=""

| "-"

| "(abs)"

| "- "(abs)"

Instruction Modification

<ConditionalModifier>::=""

| <CondMod> ". " <FlagReg>

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

<Saturate>::=""

| ".sat"

Execution Size

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

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

Instruction Options

```

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

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

Instruction Set Summary Tables

The columns in the following tables specify instruction mnemonics, hex opcodes, full names, instruction groups, processor generation (where blank means available for BDW), 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 BDW for BDW in the Project column. If the Project column is blank, an instruction is supported for BDW, all generations implemented or designed so far.

N and Y indicate No (no support for a feature) and Yes (full support for a feature) respectively.

A SrcMod (source modifier) value of Y indicates that a numeric source modifier is allowed, optionally specifying absolute value, negation, or a forced negative value. The value N indicates no source modifier support.

SrcMod Information	
A SrcMod value of ** indicates a logical source modifier is allowed, optionally inverting all source bits (a NOT operation).	

In the Src Types and Dst Type columns, Int means any integer type and * means such an extensive list of types that you must refer to the detailed instruction description.

Byte (B, UB) and QWord (Q, UQ) integer types cannot be mixed in the same instruction.

Instruction Set Summary Table A to B (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	SrCs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>add</i>	40	Addition	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>addc</i>	4E	Integer Addition with Carry	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
<i>and</i>	05	Logic And	Move and Logic	2	Y	**	Int	Int	N	Equality only
<i>asr</i>	0C	Arithmetic Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>avg</i>	42	Average	Parallel Arithmetic	2	Y	Y	B, UB W, UW D, UD	B, UB W, UW D, UD	Y	Y
<i>bfe</i>	18	Bit Field Extract	Move and Logic	3	Y	N	UD, D	UD, D	N	N
<i>bfi1</i>	19	Bit Field Insert 1	Move and Logic	2	Y	N	UD, D	UD, D	N	N
<i>bfi2</i>	1A	Bit Field Insert 2	Move and Logic	3	Y	N	UD, D	UD, D	N	N
<i>bfrev</i>	17	Bit Field Reverse	Move and Logic	1	Y	N	UD	UD	N	N
<i>brc</i>	23	Branch Converging	Flow Control	0 or 1	Y	N	D		N	N
<i>brd</i>	21	Branch Diverging	Flow Control	0 or 1	Y	N	D		N	N
<i>break</i>	28	Break	Flow Control	0	Y	N			N	N

Instruction Set Summary Table C to E (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>call</i>	2C	Call	Flow Control	0	Y	N		D, UD	N	N
<i>calla</i>	2B	Call Absolute	Flow Control	0	Y	N		D, UD	N	N
<i>cbit</i>	4D	Count Bits Set	Move and Logic	1	Y	N	UB, UW, UD	UD	N	N
<i>cmp</i>	10	Compare	Move and Logic	2	Y	Y	*	*	N	Y
<i>cmpn</i>	11	Compare NaN	Move and Logic	2	Y	Y	*	*	N	Y
<i>cont</i>	29	Continue	Flow Control	0	Y	N			N	N
<i>csel</i>	12	Conditional Select	Move and Logic	3	N	Y	F	F	Y	Y
<i>dp2</i>	57	Dot Product 2	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>dp3</i>	56	Dot Product 3	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>dp4</i>	54	Dot Product 4	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>dph</i>	55	Dot Product Homogeneous	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>else</i>	24	Else	Flow Control	0	N	N			N	N
<i>endif</i>	25	End If	Flow Control	0	N	N			N	N

Instruction Set Summary Table F to L (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>fbh</i>	4B	Find First Bit from MSB Side	Move and Logic	1	Y	N	D, UD	UD	N	N
<i>lbl</i>	4C	Find First Bit from LSB Side	Move and Logic	1	Y	N	UD	UD	N	N
<i>frc</i>	43	Fraction	Parallel Arithmetic	1	Y	Y	F	F	N	Y
<i>goto</i>	2E	Goto	Flow Control	0	Y	N			N	N
<i>halt</i>	2A	Halt	Flow Control	0	Y	N			N	N
<i>if</i>	22	If	Flow Control	0	Y	N			N	N
<i>illegal</i>	00	Illegal	Special	0	N	N			N	N
<i>jmp</i>	20	Jump Indexed	Flow Control	1	Y	N	D		N	N
<i>join</i>	2F	Join	Flow Control	0	Y	N			N	N
<i>line</i>	59	Line	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>lrp</i>	5C	Linear Interpolation	Vector Arithmetic	3	Y	Y	F	F	N	Y
<i>lzd</i>	4A	Leading Zero Detection	Move and Logic	1	Y	Y	D, UD	UD	Y	Y

Instruction Set Summary Table M to P (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>mac</i>	48	Multiply Accumulate	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>mach</i>	49	Multiply Accumulate High	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>mad</i>	5B	Multiply Add	Vector Arithmetic	3	Y	Y	*	*	Y	Y
<i>madm</i>	5D	Multiply Add for Macro	Vector Arithmetic	3	Y	Y	*	*	Y	Y
<i>math</i>	38	Extended Math Function	Miscellaneous	2	Y	N	*	*	Y	N
<i>mov</i>	01	Move	Move and Logic	1	Y	Y	*	*	Y	Y
<i>movi</i>	03	Move Indexed	Move and Logic	1	Y	Y	*	*	Y	N
<i>mul</i>	41	Multiply	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>nop</i>	7E	No Operation	Special	0	N	N			N	N
<i>not</i>	04	Logic Not	Move and Logic	1	Y	**	Int	Int	N	Equality only
<i>or</i>	06	Logic Or	Move and Logic	2	Y	**	Int	Int	N	Equality only
<i>pln</i>	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?
<i>ret</i>	2D	Return	Flow Control	1	Y	N	D, UD		N	N
<i>rndd</i>	45	Round Down	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rnde</i>	46	Round to Nearest or Even	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rndu</i>	44	Round Up	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rndz</i>	47	Round to Zero	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>sad2</i>	50	Sum of Absolute Difference 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
<i>sada2</i>	51	Sum of Absolute Difference Accumulate 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
<i>sel</i>	02	Select	Move and Logic	2	Y	Y	*	*	Y	Y
<i>send</i>	31	Send Message	Miscellaneous	1	Y	N	*	*	N	N
<i>sendc</i>	32	Conditional Send Message	Miscellaneous	1	Y	N	*	*	N	N
<i>shl</i>	09	Shift Left	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>shr</i>	08	Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>smov</i>	0A	Scattered Move	Move and Logic	1	Y	N	*	*	N	N
<i>subb</i>	4F	Integer Subtraction with Borrow	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
<i>wait</i>	30	Wait	Miscellaneous	1	N	N	UD	UD	N	N
<i>while</i>	27	While	Flow Control	0	Y	N			N	N
<i>xor</i>	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 with three source operands cannot use explicit accumulator operands. AccWrEn may be allowed for implicitly updating the accumulator.
3. Instructions that use the accumulator as an implicit source value cannot specify an explicit accumulator source operand.
4. Instructions that specify an implicit accumulator destination (with AccWrEn) cannot specify an explicit accumulator destination operand.
5. 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.

These descriptions are frequently used in this table:

- No restrictions.
- No accumulator access, implicit or explicit.
- Source operands cannot be accumulators.
- Source modifier is not allowed if source is an accumulator.
- Accumulator is an implicit source and thus cannot be an explicit source operand.
- Accumulator cannot be destination, implicit or explicit.
- AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

These minor cases occur occasionally in the table:

- Integer source operands cannot be accumulators.
- No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
- An accumulator can be a source or destination operand but not both.

A few instructions use more than one of the listed restrictions.

Accumulator Restrictions

Instructions	Accumulator Restrictions
<i>add</i>	No restrictions.
<i>addc</i>	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
<i>and</i>	Source modifier is not allowed if source is an accumulator.
<i>asr</i> <i>avg</i>	No restrictions.
<i>bfe</i> <i>bfi1</i> <i>bfi2</i> <i>bfrex</i> <i>cbit</i>	No accumulator access, implicit or explicit.
<i>cmp</i>	No restrictions.
<i>cmpn</i>	No restrictions.
<i>csl</i>	No restrictions.
<i>dp2</i> <i>dp3</i> <i>dp4</i> <i>dph</i>	Source operands cannot be accumulators.
<i>fbh</i> <i>fbl</i>	No accumulator access, implicit or explicit.
<i>frc</i>	No restrictions.
<i>line</i>	Source operands cannot be accumulators.
<i>lrp</i>	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
<i>lzd</i>	No restrictions.
<i>mac</i>	Accumulator is an implicit source and thus cannot be an explicit source operand.
<i>mach</i>	Accumulator is an implicit source and thus cannot be an explicit source operand. AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
<i>madm</i>	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
<i>math</i>	No accumulator access, implicit or explicit.
<i>mov</i>	An accumulator can be a source or destination operand but not both.
<i>movi</i>	Source operands cannot be accumulators.
<i>mul</i>	Integer source operands cannot be accumulators.
<i>not</i> <i>or</i>	Source modifier is not allowed if source is an accumulator.
<i>pln</i>	Source operands cannot be accumulators.

Instructions	Accumulator Restrictions
<i>rndd</i> <i>rnde</i> <i>rndu</i> <i>rndz</i>	No restrictions.
<i>sad2</i> <i>sada2</i>	Source operands cannot be accumulators.
<i>sel</i>	No restrictions.
<i>shl</i>	Accumulator cannot be destination, implicit or explicit.
<i>shr</i>	No restrictions.
<i>smov</i>	No restrictions.
<i>subb</i>	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
<i>xor</i>	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
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 <ul style="list-style-type: none"> • INV - Inverse • LOG – Logarithm • EXP - Exponent • SQRT - Square Root • RSQ - Reciprocal Square Root • POW - Power Function • SIN - SINE • COS - COSINE • INT DIV - Integer Divide • INVM/RSQRTM 	EUISA
fbl	Find First Bit from LSB Side	EUISA

Symbol	Name	Source
fbh	Find First Bit from MSB Side	EUIA
frc	Fraction	EUIA
goto	Goto	EUIA
halt	Halt	EUIA
if	If	EUIA
illebal	Illegal	EUIA
subb	Integer Subtraction with Borrow	EUIA
join	Join	EUIA
jmp	Jump Indexed	EUIA
lzd	Leading Zero Detection	EUIA
line	Line	EUIA
lrp	Linear Interpolation	EUIA
and	Logic And	EUIA
not	Logic Not	EUIA
or	Logic Or	EUIA
xor	Logic Xor	EUIA
mov	Move	EUIA
movi	Move Indexed	EUIA
mul	Multiply	EUIA
mac	Multiply Accumulate	EUIA
mach	Multiply Accumulate High	EUIA
mad	Multiply Add	EUIA
madm	Multiply Add for Macro	EUIA
nop	No Operation	EUIA
pln	Plane	EUIA
ret	Return	EUIA
rndd rnde rndu rndz	Round Instructions <ul style="list-style-type: none"> ▪ Round Down ▪ Round to Nearest or Even ▪ Round Up ▪ Round to Zero 	EUIA
smov	Scattered Move	EUIA
sel	Select	EUIA
send	Send Message	EUIA
shl	Shift Left	EUIA
shr	Shift Right	EUIA
sad	Sum of Absolute Difference 2	EUIA
sada2	Sum of Absolute Difference Accumulate 2	EUIA

Symbol	Name	Source
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:									
^	Project	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 <i>src0</i> is {–inf, +inf, NaN}.								

rnde – Round to Nearest or Even

Description:

The *rnde* instruction takes component-wise floating point round-to-even operation of *src0* with results in two pieces – a downward rounded integral float results stored in *dst* and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in *dst* to create the final round-to-even values to emulate the round-to-even operation, commonly known as the *round()* function. The final results are the one of the two integral float values that is nearer to the input values. If the neither possibility is nearer, the even alternative is chosen.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round to Nearest or Even in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	*	–0	–0	+0	+0	**	+inf	NaN
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								

Floating-Point Round to Nearest or Even in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	**	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if <i>src0</i> is {–inf, +inf, NaN}.								

rndu – Round Up

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of *src0*, commonly known as the *ceil*ing() 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}.								
^	<div style="border: 1px solid black; padding: 5px; text-align: center;"> Note {+1, +0} depending on the Single Precision Denorm Mode. </div>								

Floating-Point Round Up in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	+finite	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
***	Result is undefined if <i>src0</i> is {–inf, +inf, NaN}.								

rndz – Round to Zero

Description:

The *rndz* instruction takes component-wise floating point round-to-zero operation of *src0* with results in two pieces – a downward rounded integral float results stored in *dst* and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in *dst* to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the *truncate()* function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.

Floating-Point Round to Zero in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	*	–0	–0	+0	+0	**	+inf	NaN
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								

Floating-Point Round to Zero in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	**	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if <i>src0</i> is {–inf, +inf, NaN}.								

math – Extended Math Function

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 $\text{Log}_2(\text{abs}(\text{src0}))$

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 $\text{SQRT}(\text{abs}(\text{src0}))$

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 $\text{RSQ}(\text{abs}(\text{src0}))$

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 $\pm 100 \cdot \pi$

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

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 $\pm 100 \times \pi$

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) r11.acc5 r8.acc2 r10.acc4 r8.acc2 // Step(3), y1=y0+e0*y0
madm (8) r11.acc6 r6.noacc -r4.noacc r9.acc3 // Step(4), r0=a-b*q0
madm (8) r9.acc7 r9.acc3 r11.acc6 r11.acc5 // Step(5), q1=q0+r0*y1
madm (8) r6.acc8 r6.noacc -r4.noacc r9.acc7 // Step(6), r1=a-b*q1

Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL

madm (8) r8.noacc r9.acc7 r6.acc8 r11.acc5 // Step(7), q=q1+r1*y1
endif
```

Macro for Double Precision IEEE Compliant fdiv

```
Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r6 = a, r7 = b, r1 = 1
The default data type for the macro is :df

math.eo.f0.0 (4) r8.acc2 r6.noacc r7.noacc 0xE
(-f0.0) if
madm (4) r9.acc3 r0.noacc r6.noacc r8.acc2 // Step(1), q0=a*y0
madm (4) r10.acc4 r1.noacc -r7.noacc r8.acc2 // Step(2), e0=(1-b*y0)
madm (4) r11.acc5 r6.noacc -r7.noacc r9.acc3 // Step(3), r0=a-b*q0
madm (4) r12.acc6 r8.acc2 r10.acc4 r8.acc2 // Step(4), y1=y0+e0*y0
madm (4) r13.acc7 r1.noacc -r7.noacc r12.acc6 // Step(5), e1=(1-b*y1)
madm (4) r8.acc8 r8.acc2 r10.acc4 r12.acc6 // Step(6), y2=y0+e0*y1
madm (4) r9.acc9 r9.acc3 r11.acc5 r12.acc6 // Step(7), q1=q0+r0*y1
madm (4) r12.acc2 r12.acc6 r8.acc8 r13.acc7 // Step(8), y3=y1+e1*y2
madm (4) r11.acc3 r6.noacc -r7.noacc r9.acc9 // Step(9), r1=a-b*q1
```

Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL

```
madm (4) r8.noacc r9.acc9 r11.acc3 r12.acc2 // Step(10), q=q1+r1*y3
endif
```

Macro for Single Precision IEEE Compliant sqrt

Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r8 = 1/2, r6 = a
The default datatype for the macro is :f

```
math.eo.f0.0 (8) r7.acc2 r6.noacc null 0xF
(-f0.0) if
madm (8) r9.acc3 r0.noacc r8.noacc r7.acc2 // Step(1), H0=1/2*y0
madm (8) r11.acc4 r0.noacc r6.noacc r7.acc2 // Step(2), S0=a*y0
madm (8) r10.acc5 r8.noacc -r11.acc4 r9.acc3 // Step(3), d0=1/2-S0*H0
madm (8) r9.acc6 r9.acc3 r10.acc5 r9.acc3 // Step(4), H1=H0+d0*H0
madm (8) r7.acc7 r11.acc4 r10.acc5 r11.acc4 // Step(5), S1=S0+d0*S0
madm (8) r10.acc8 r6.noacc -r7.acc7 r7.acc7 // Step(6), e0=a-S1*S1
```

Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL

```
madm (8) r7.noacc r7.acc7 r9.acc6 r10.acc8 // Step(7), S=S1+e0*H1
endif
```

Macro for Double Precision IEEE Compliant sqrt

Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r8 = 1/2, r6 = a, r1 = 1
The default datatype for the macro is :df

```
math.eo.f0.0 (4) r7.acc2 r6.noacc null 0xF
(-f0.0) if
madm (4) r9.acc3 r0.noacc r8.noacc r7.acc2 // Step(1), H0=1/2*y0
madm (4) r11.acc4 r0.noacc r6.noacc r7.acc2 // Step(2), S0=a*y0
madm (4) r10.acc5 r8.noacc -r11.acc4 r9.acc3 // Step(3), d=1/2-S0*H0
madm (4) r8.acc6 r1.noacc r8.noacc r1.noacc // Step(0'') 3/2=1+1/2
madm (4) r8.acc7 r1.noacc r8.acc6 r10.acc5 // Step(4), e=1+3/2*d
madm (4) r7.acc8 r0.noacc r10.acc5 r11.acc4 // Step(5), T0=d*S0
madm (4) r10.acc9 r0.noacc r10.acc5 r9.acc3 // Step(6), G0=d*H0
madm (4) r7.acc8 r11.acc4 r8.acc7 r7.acc8 // Step(7), S1=S0+e*T0
madm (4) r8.acc7 r9.acc3 r8.acc7 r10.acc9 // Step(8), H1=H0+e*G0
madm (4) r9.acc3 r6.noacc -r7.acc8 r7.acc8 // Step(9), d1=a-S1*S1
```

Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL

```
madm (4) r7.noacc r7.acc8 r9.acc3 r8.acc7 // Step(10), S=S1+d1*H1
endif
```

Restrictions for the Use of Macros

1. Macro Operations need to be used as special subroutines. Interleaving other operations is not recommended.
2. All operations are in Align16 mode.

3. Single precision operations must be SIMD8 and double precision operations must be SIMD4.
4. Channel Select and Channel Enable controls are re-purposed for implicit accumulator controls. Hence the default swizzle is always .xyzw and all destination channels are enabled.
5. The macro sequences should be used with the Rounding Mode in CR set to RNE. The Rounding Mode may be changed only for the last *madm* of the macro depending on the requirement.
6. Macro Operations are not supported.

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:

```
[(pred)] send (exec_size) reg greg imm6 reg32a
[(pred)] send (exec_size) reg greg imm6 imm32
```

Pseudocode:

```
Evaluate(WrEn);
<MsgChEnable> = WrEn;
<SourceReg> = <src>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>, <ex_desc>, <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 message descriptor field <ex_desc> contains the target function ID. WrEn is forwarded to the target function in the 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.

<ex_desc> is a 6-bit immediate, *imm6*. The lower 4bits of the <ex_desc> specifies the SFID for the message. The MSb of the message descriptor, the EOT field, always comes from bit 127 of the instruction word, which is the MSb of *imm6*. A thread must terminate with a *send* instruction with EOT turned on.

<src> is a 256-bit aligned GRF register. It serves as the leading GRF register of the request.

<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

Bit	Description
31	Reserved : MBZ
30	Reserved : MBZ
29	Reserved : MBZ
28:25	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]
24:20	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]
19	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
18:0	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	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. <i>Refer to "GPU Overview" document for the mapping of Shared Function IDs</i>

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	EuIsa
DstRegNum	EuIsa
DstSubRegNum	EuIsa
EU_DEBUG_REGISTERS	PRM
EU_INSTRUCTION_BASIC_ONE_SRC	EuIsa
EU_INSTRUCTION_BASIC_THREE_SRC	EuIsa
EU_INSTRUCTION_BASIC_TWO_SRC	EuIsa
EU_INSTRUCTION_BRANCH_CONDITIONAL	EuIsa
EU_INSTRUCTION_BRANCH_ONE_SRC	EuIsa
EU_INSTRUCTION_BRANCH_TWO_SRC	EuIsa
EU_INSTRUCTION_COMPACT_THREE_SRC	EuIsa
EU_INSTRUCTION_COMPACT_TWO_SRC	EuIsa
EU_INSTRUCTION_CONTROLS	EuIsa
EU_INSTRUCTION_CONTROLS_A	EuIsa
EU_INSTRUCTION_CONTROLS_B	EuIsa
EU_INSTRUCTION_HEADER	EuIsa
EU_INSTRUCTION_ILLEGAL	EuIsa
EU_INSTRUCTION_MATH	EuIsa
EU_INSTRUCTION_NOP	EuIsa
EU_INSTRUCTION_OPERAND_CONTROLS	EuIsa
EU_INSTRUCTION_OPERAND_DST_ALIGN1	EuIsa
EU_INSTRUCTION_OPERAND_DST_ALIGN16	EuIsa
EU_INSTRUCTION_OPERAND_SEND_MSG	EuIsa
EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN1	EuIsa
EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN16	EuIsa
EU_INSTRUCTION_OPERAND_SRC_REG_THREE_SRC	EuIsa
EU_INSTRUCTION_SEND	EuIsa
EU_INSTRUCTION_SOURCES_IMM32	EuIsa
EU_INSTRUCTION_SOURCES_REG	EuIsa
EU_INSTRUCTION_SOURCES_REG_IMM	EuIsa
EU_INSTRUCTION_SOURCES_REG_REG	EuIsa
ExtMsgDescpt	EuIsa
FunctionControl	EuIsa
MsgDescpt31	EuIsa
SrcRegNum	EuIsa
SrcSubRegNum	EuIsa

EUIA Enumerations

Name	Source
AddrMode	EuIsa
ChanEn	EuIsa
ChanSel	EuIsa
CondModifier	EuIsa
DepCtrl	EuIsa
DstType	EuIsa
EU_OPCODE	EuIsa
ExecSize	EuIsa
FC	EuIsa
HorzStride	EuIsa
PredCtrl	EuIsa
QtrCtrl	EuIsa
RegFile	EuIsa
RepCtrl	EuIsa
SFID	EuIsa
SrcImmType	EuIsa
SrcIndex	EuIsa
SrcMod	EuIsa
SrcType	EuIsa
ThreadCtrl	EuIsa
VertStride	EuIsa
Width	EuIsa

EU Programming Guide

Assembler Pragma 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
<ul style="list-style-type: none"> .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 Cartesian 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 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx// r1
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx// r2 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx// r3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

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 Cartesian 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 00000000// 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

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`

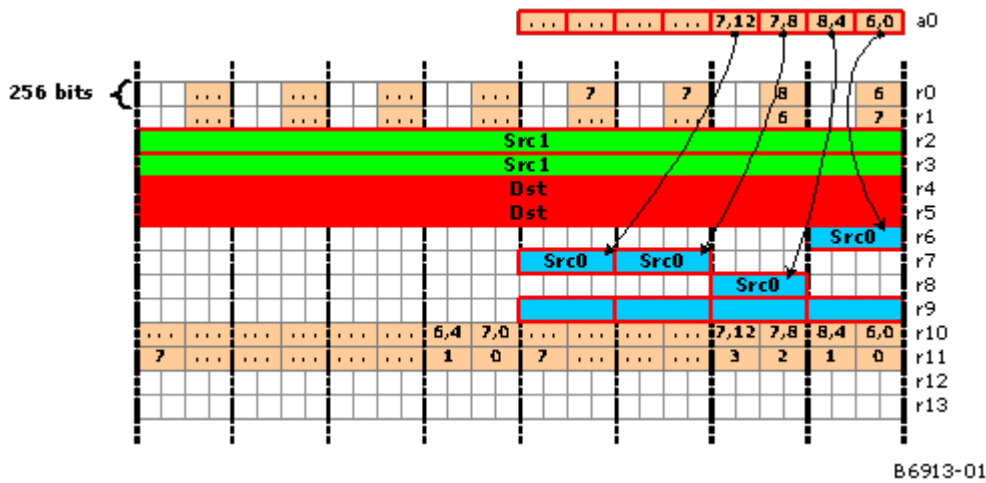
`mov (8)a0.0<1>:wr10.0<0;8,1>:w`

`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

SWZr1r0.0-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>:fr0.xzxz0x1F111F11:v// Constant vector of (1 -1 1 1 1 -1 1 1)

mov (1)f0.08b'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:

SWZr1r0.0zx-1// Expected results: r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1

GEN:

mov (1)f0.08b'01100110// Set flag and masked out channels x and w

(f0.0)sel (8) r1.0<1>:f r0.yzxy0x000F000F: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 {NoDDCChk}
```

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, Co#) 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: $(X0 \cdot Cx0 + Y0 \cdot Cy0 + Co0) \dots (X7 \cdot Cx0 + Y7 \cdot Cy0 + Co0)$

R5: $(X0 \cdot Cx1 + Y0 \cdot Cy1 + Co1) \dots (X7 \cdot Cx1 + Y7 \cdot Cy1 + Co1)$

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 $acc = X\# \cdot Cx0 + Co0$

mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does $r4.\# = Y\# \cdot Cy0 + acc.\#$

line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f// does $acc = X\# \cdot Cx0 + Co0$

mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does $r4.\# = Y\# \cdot Cy0 + acc.\#$

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: $(X0 * Cx0 + Y0 * Cy0 + Z0 * Cz0 + Co0) \dots (X7 * Cx0 + Y7 * Cy0 + Z7 * Cz0 + Co0)$

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 access, 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 absolute
add (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) IP 4*16// jumps to HERE_1
add (1) IP 0x35// 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 loop
add (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

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// r2 > 0x80?
cmp.g.f3 (16) null r3 0x80// r3 > 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.


```

<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 jmpri to perform a case statement  
// of any number of options in 3 jumps  
//-----  
.default_execution_size 8  
...  
jmpri 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...
```