Copyright

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 © 2013, Intel Corporation. All rights reserved.
## 3D-Media-GPGPU Engine

### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render Engine Command Memory Interface</td>
<td>37</td>
</tr>
<tr>
<td>Registers in Render Engine</td>
<td>38</td>
</tr>
<tr>
<td>Mode and Misc Ctrl Registers</td>
<td>39</td>
</tr>
<tr>
<td>Pipelines Statistics Counter Registers</td>
<td>40</td>
</tr>
<tr>
<td>Predicate Render Registers</td>
<td>41</td>
</tr>
<tr>
<td>AUTO_DRAW Registers</td>
<td>42</td>
</tr>
<tr>
<td>MMIO Registers for GPGPU Indirect Dispatch</td>
<td>43</td>
</tr>
<tr>
<td>CS ALU</td>
<td>44</td>
</tr>
<tr>
<td>ALU PROGRAMMING</td>
<td>44</td>
</tr>
<tr>
<td>ALU DESIGN</td>
<td>44</td>
</tr>
<tr>
<td>Generic Purpose Registers</td>
<td>44</td>
</tr>
<tr>
<td>ALU BLOCK Diagram</td>
<td>45</td>
</tr>
<tr>
<td>Instruction Set</td>
<td>45</td>
</tr>
<tr>
<td>Instruction Format</td>
<td>46</td>
</tr>
<tr>
<td>LOAD Operation</td>
<td>46</td>
</tr>
<tr>
<td>Arithmetic/Logical Operations</td>
<td>47</td>
</tr>
<tr>
<td>STORE Operation</td>
<td>48</td>
</tr>
<tr>
<td>Summary for ALU</td>
<td>48</td>
</tr>
<tr>
<td>Summary of Instructions Supported</td>
<td>49</td>
</tr>
<tr>
<td>Table for ALU OPCODE Encodings</td>
<td>50</td>
</tr>
<tr>
<td>Table for Register Encodings</td>
<td>51</td>
</tr>
<tr>
<td>CS_GPR - Command Streamer General Purpose Registers</td>
<td>52</td>
</tr>
<tr>
<td>Memory Interface Commands for Rendering Engine</td>
<td>53</td>
</tr>
<tr>
<td>Predicated Rendering Support in HW</td>
<td>54</td>
</tr>
<tr>
<td>MI_SET_PREDICATE</td>
<td>55</td>
</tr>
<tr>
<td>State Commands</td>
<td>56</td>
</tr>
<tr>
<td>STATE_BASE_ADDRESS</td>
<td>56</td>
</tr>
<tr>
<td>Synchronization of the 3D Pipeline</td>
<td>57</td>
</tr>
<tr>
<td>Top-of-Pipe Synchronization</td>
<td>57</td>
</tr>
</tbody>
</table>
3DSTATE_GATHER_DS .................................................................................................................................. 100
3DSTATE_GATHER_CONSTANT_GS ............................................................................................................... 101
3DSTATE_GATHER_CONSTANT_PS .............................................................................................................. 103
Dx9 Constant Buffer Generation ................................................................................................................ 104
Vertex Shader Constant ............................................................................................................................... 105
Pixel Shader Constant ................................................................................................................................... 106

Shared Functions .................................................................................................................................. 107
3D Sampler .................................................................................................................................................. 107
Sampling Engine ....................................................................................................................................... 109
Texture Coordinate Processing .............................................................................................................. 111
Texture Coordinate Normalization ........................................................................................................ 111
Texture Coordinate Computation ........................................................................................................... 111
Texel Address Generation ........................................................................................................................ 113
Level of Detail Computation (Mipmapping) .......................................................................................... 113
Base Level Of Detail (LOD) ..................................................................................................................... 114
LOD Bias ................................................................................................................................................... 114
LOD Pre-Clamping .................................................................................................................................... 114
Min/Mag Determination .......................................................................................................................... 115
LOD Computation Pseudocode ............................................................................................................. 116
Inter-Level Filtering Setup .................................................................................................................... 117
Intra-Level Filtering Setup ...................................................................................................................... 118
MAPFILTER_NEAREST ............................................................................................................................ 119
MAPFILTER_LINEAR ................................................................................................................................. 119
MAPFILTER_ANISOTROPIC ...................................................................................................................... 120
MAPFILTER_MONO ................................................................................................................................... 121
Texture Address Control ............................................................................................................................ 123
TEXCOORDMODE_MIRROR Mode ............................................................................................................. 124
TEXCOORDMODE_WRAP Mode ................................................................................................................... 124
TEXCOORDMODE_MIRROR_ONCE Mode ................................................................................................. 124
TEXCOORDMODE_CLAMP Mode .................................................................................................................. 125
TEXCOORDMODE_CLAMPBORDER Mode ............................................................................................... 125
TEXCOORDMODE_CUBE Mode .................................................................................................................... 125
Texel Fetch ............................................................................................................................................... 126
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Payload (Write)</td>
<td>255</td>
</tr>
<tr>
<td>Message Payload (Read)</td>
<td>255</td>
</tr>
<tr>
<td>Writeback Message (Read)</td>
<td>255</td>
</tr>
<tr>
<td>Memory Fence</td>
<td>256</td>
</tr>
<tr>
<td>Message Descriptor</td>
<td>257</td>
</tr>
<tr>
<td>Message Header</td>
<td>257</td>
</tr>
<tr>
<td>Writeback Message</td>
<td>257</td>
</tr>
<tr>
<td>Pixel Data Port</td>
<td>258</td>
</tr>
<tr>
<td>Cache Agents</td>
<td>258</td>
</tr>
<tr>
<td>Accessing Render Targets</td>
<td>258</td>
</tr>
<tr>
<td>Message Sequencing Summary</td>
<td>258</td>
</tr>
<tr>
<td>Single Source</td>
<td>260</td>
</tr>
<tr>
<td>Dual Source</td>
<td>260</td>
</tr>
<tr>
<td>Replicate Data</td>
<td>261</td>
</tr>
<tr>
<td>Multiple Render Targets (MRT)</td>
<td>261</td>
</tr>
<tr>
<td>Render Target Read and Write</td>
<td>261</td>
</tr>
<tr>
<td>Subspan/Pixel to Slot Mapping</td>
<td>264</td>
</tr>
<tr>
<td>Message Descriptor</td>
<td>266</td>
</tr>
<tr>
<td>Message Header</td>
<td>266</td>
</tr>
<tr>
<td>Message Header</td>
<td>266</td>
</tr>
<tr>
<td>Writeback Message (Read)</td>
<td>267</td>
</tr>
<tr>
<td>Header for SIMD8_IMAGE_WRITE</td>
<td>269</td>
</tr>
<tr>
<td>Source 0 Alpha Payload</td>
<td>272</td>
</tr>
<tr>
<td>oMask Payload</td>
<td>273</td>
</tr>
<tr>
<td>Color Payload: SIMD16 Single Source</td>
<td>274</td>
</tr>
<tr>
<td>Color Payload</td>
<td>274</td>
</tr>
<tr>
<td>Color Payload: SIMD8 Single Source</td>
<td>275</td>
</tr>
<tr>
<td>Color Payload SIMD16 Replicated Data</td>
<td>276</td>
</tr>
<tr>
<td>Color Payload SIMD8 Dual Source</td>
<td>276</td>
</tr>
<tr>
<td>Message Sequencing Summary</td>
<td>278</td>
</tr>
<tr>
<td>Message Sequencing Summary</td>
<td>278</td>
</tr>
<tr>
<td>Render Target Read and Write</td>
<td>279</td>
</tr>
<tr>
<td>Message Header</td>
<td>282</td>
</tr>
</tbody>
</table>
CloseGateway Message........................................................................................................................... 335
Message Payload................................................................................................................................... 335
Writeback Message to Requester Thread ............................................................................................. 335
ForwardMsg Message.............................................................................................................................. 336
Message Payload................................................................................................................................... 336
Writeback Message to Requester Thread ............................................................................................. 338
Writeback Message to Recipient Thread ............................................................................................. 339
GetTimeStamp Message.......................................................................................................................... 339
Message Payload................................................................................................................................... 339
Writeback Message to Requester Thread ............................................................................................. 340
BarrierMsg Message................................................................................................................................. 341
Message Payload................................................................................................................................... 341
Writeback Message to Requester Thread ............................................................................................. 342
Broadcast Writeback Message .............................................................................................................. 342
MMIOReadWrite Message......................................................................................................................... 343
Message Payload................................................................................................................................... 343
Writeback Message to Requester Thread (MMIO Read Only) ............................................................. 343

Shared Functions - Media Sampler ......................................................................................................... 345

Video Motion Estimation ....................................................................................................................... 346

Theory of Operation ............................................................................................................................... 346

Shape Decision ....................................................................................................................................... 346

Minor Shape Decision Prior to FME ..................................................................................................... 347
Major Shape Decision Prior to FME ..................................................................................................... 350
Shape Update after FME ....................................................................................................................... 350
Final Code Decision after BME ........................................................................................................... 350

Integer Motion Estimation ................................................................................................................... 351
Reference Window and Search Units ..................................................................................................... 351
Fixed and Adaptive Search Paths ........................................................................................................ 353

Fractional Motion Estimation ............................................................................................................... 357
Interpolations ........................................................................................................................................ 357
8+8 vs. 7x7 ........................................................................................................................................... 358
Partitioning Refinement ....................................................................................................................... 358

BME and Weighted Prediction ........................................................................................................... 359
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>499</td>
</tr>
<tr>
<td>Functions</td>
<td>500</td>
</tr>
<tr>
<td>SIMD4x2 Thread Execution</td>
<td>500</td>
</tr>
<tr>
<td>Statistics Gathering</td>
<td>500</td>
</tr>
<tr>
<td>Payloads</td>
<td>501</td>
</tr>
<tr>
<td>SIMD4x2 Payload</td>
<td>501</td>
</tr>
<tr>
<td>3D Pipeline – Geometry Shader (GS) Stage</td>
<td>505</td>
</tr>
<tr>
<td>GS Stage Overview</td>
<td>505</td>
</tr>
<tr>
<td>State</td>
<td>505</td>
</tr>
<tr>
<td>Functions</td>
<td>506</td>
</tr>
<tr>
<td>Object Staging</td>
<td>506</td>
</tr>
<tr>
<td>Thread Request Generation</td>
<td>506</td>
</tr>
<tr>
<td>Object Vertex Ordering</td>
<td>510</td>
</tr>
<tr>
<td>Thread Execution</td>
<td>512</td>
</tr>
<tr>
<td>GS URB Entry</td>
<td>512</td>
</tr>
<tr>
<td>GS Output Topologies</td>
<td>513</td>
</tr>
<tr>
<td>GS Output StreamID</td>
<td>513</td>
</tr>
<tr>
<td>Primitive Output</td>
<td>514</td>
</tr>
<tr>
<td>Statistics Gathering</td>
<td>514</td>
</tr>
<tr>
<td>GS Invocations</td>
<td>514</td>
</tr>
<tr>
<td>Payloads</td>
<td>514</td>
</tr>
<tr>
<td>Thread Payload High-Level Layout</td>
<td>514</td>
</tr>
<tr>
<td>SIMD 4x2 Thread Payload</td>
<td>516</td>
</tr>
<tr>
<td>Thread Control Information</td>
<td>524</td>
</tr>
<tr>
<td>Thread Payload Generation</td>
<td>525</td>
</tr>
<tr>
<td>Fixed Payload Header</td>
<td>526</td>
</tr>
<tr>
<td>Extended Payload Header</td>
<td>528</td>
</tr>
<tr>
<td>Payload URB Data</td>
<td>528</td>
</tr>
<tr>
<td>3D Pipeline - Stream Output Logic (SOL) Stage</td>
<td>530</td>
</tr>
<tr>
<td>State</td>
<td>530</td>
</tr>
<tr>
<td>Functions</td>
<td>531</td>
</tr>
<tr>
<td>Input Buffering</td>
<td>531</td>
</tr>
</tbody>
</table>
Stream Output Function ..................................................................................................................... 534
Stream Output Buffers........................................................................................................................ 535
Rendering Disable .............................................................................................................................. 535
Statistics .................................................................................................................................................... 536
3D Pipeline Rasterization .................................................................................................................... 537
3D Pipeline – CLIP Stage Overview ........................................................................................................ 537
Clip Stage – General-Purpose Processing .......................................................................................... 537
Clip Stage – 3D Clipping ...................................................................................................................... 537
Fixed Function Clipper ......................................................................................................................... 538
Concepts .................................................................................................................................................. 538
The Clip Volume .................................................................................................................................... 538
View Volume ....................................................................................................................................... 538
User-Specified Clipping ...................................................................................................................... 540
Guard Band .............................................................................................................................................. 540
NDC Guardband Parameters ....................................................................................................... 542
Vertex-Based Clip Testing Considerations .................................................................................. 543
   Triangle Objects ................................................................................................................................ 543
   Non-Wide Line Objects .................................................................................................................. 544
   Wide Line Objects ............................................................................................................................ 544
   Wide Points ......................................................................................................................................... 544
   RECTLIST ............................................................................................................................................... 545
3D Clipping ........................................................................................................................................... 545
CLIP Stage Input ......................................................................................................................................... 545
State ............................................................................................................................................................ 546
VUE Readback ............................................................................................................................................. 546
VertexClipTest Function .......................................................................................................................... 546
Object Staging ........................................................................................................................................... 551
Partial Object Removal ....................................................................................................................... 552
ClipDetermination Function .............................................................................................................. 552
ClipMode .................................................................................................................................................. 555
   NORMAL ClipMode .......................................................................................................................... 556
   CLIP_ALL ClipMode .......................................................................................................................... 556
   CLIP_NON_REJECT ClipMode ............................................................................................................ 556
REJECT_ALL ClipMode..................................................................................................................... 556
ACCEPT_ALL ClipMode................................................................................................................... 556
Object Pass-Through....................................................................................................................... 557
Primitive Output............................................................................................................................ 558
Other Functionality......................................................................................................................... 558
Statistics Gathering......................................................................................................................... 558
CL_INVOCATION_COUNT.............................................................................................................. 558
3D Pipeline - Strips and Fans (SF) Stage.......................................................................................... 559
Inputs from CLIP.............................................................................................................................. 559
Attribute Setup/Interpolation Process............................................................................................ 560
Attribute Setup/Interpolation Process............................................................................................ 560
Outputs to WM................................................................................................................................... 560
Primitive Assembly.......................................................................................................................... 560
Point List Decomposition............................................................................................................... 564
Line List Decomposition.................................................................................................................. 565
Line Strip Decomposition................................................................................................................ 566
Triangle List Decomposition............................................................................................................ 567
Triangle Strip Decomposition......................................................................................................... 568
Triangle Fan Decomposition........................................................................................................... 569
Polygon Decomposition.................................................................................................................. 571
Rectangle List Decomposition....................................................................................................... 571
Object Setup.................................................................................................................................... 573
Invalid Position Culling (Pre/Post-Transform) ............................................................................. 573
Viewport Transformation.................................................................................................................. 573
Destination Origin Bias................................................................................................................... 573
Point Rasterization Rule Adjustment............................................................................................... 574
Drawing Rectangle Offset Application............................................................................................ 575
Point Width Application................................................................................................................... 576
Rectangle Completion...................................................................................................................... 577
Vertex XY Clamping and Quantization............................................................................................. 578
Degenerate Object Culling............................................................................................................... 578
Triangle Orientation (Face) Culling.................................................................................................. 579
Scissor Rectangle Clipping................................................................................................................ 580
Pre-Blend Color Clamping When Blending is Disabled .................................................... 643
Pre-Blend Color Clamping When Blending is Enabled....................................................... 643
Color Buffer Blending ......................................................................................................... 643
Post-Blend Color Clamping ................................................................................................. 646
Dithering .............................................................................................................................. 646
Logic Ops ............................................................................................................................. 647
Buffer Update ...................................................................................................................... 648
Stencil Buffer Updates ....................................................................................................... 648
Depth Buffer Updates ........................................................................................................ 649
Color Gamma Correction ................................................................................................. 649
Color Buffer Updates .......................................................................................................... 650
Pixel Pipeline State Summary ............................................................................................ 650
COLOR_CALC_STATE ........................................................................................................... 650
3DSTATE_BLEND_STATE_POINTERS ............................................................................... 650
3DSTATE_DEPTH_STENCIL_STATE_POINTERS .............................................................. 650
COLOR_CALC_STATE .......................................................................................................... 650
DEPTH_STENCIL_STATE ..................................................................................................... 650
BLEND_STATE ................................................................................................................... 650
CC_VIEWPORT .................................................................................................................. 650
Other Pixel Pipeline Functions .......................................................................................... 650
Statistics Gathering ........................................................................................................... 650
MCS Buffer for Render Target(s) ...................................................................................... 651
Render Target Fast Clear .................................................................................................... 654
Render Target Resolve ....................................................................................................... 654
L3/URB ............................................................................................................................... 656
L3$/URB .............................................................................................................................. 656
L3$ Cache Configuration .................................................................................................... 657
Memory Object Control State on Cacheability ................................................................. 657
Atomics ............................................................................................................................... 657
Atomics in L3 ...................................................................................................................... 660
Atomics in SLM .................................................................................................................. 660
Atomics in URB .................................................................................................................. 660
L3 Allocation & Programming .......................................................................................... 660
Non-SLM Mode Allocation

SLM Mode Allocation

L3 Invalidation and Flush Flows

Read Only Stream Invalidations

Pipelined Flush for Writes

Global Invalidation

Shared Local Memory (SLM)

Dynamic Parity Feature for GFX L3 Cache

Feature Definition

Hardware and Software Flows

Parity Generation & Detection

Correction Using Parity Error data and Redundant Rows

Number of Corrections

Summary

Sub-banks with more than two persistent parity error rows

Interrupt Enabling

Clearing the Error Reporting Registers

Software Requirement on Silent Data Corruptions

Hardware Registers

Error Report Registers

L3CDERRST1 - L3CD Error Status Register 1

Row Replacement Registers

L3B0REG0 - L3 bank0 reg0 log error

L3 Register Space

SARERRST0 - SARB Error Status slice0

L3CDERRST01 - L3CD Error Status register 1 slice 0

L3CDERRST02 - L3CD Error Status register 2 slice 0

L3SQCREG1 - L3 SQC registers 1

L3SQCREG2 - L3 SQC registers 2

L3SQCREG3 - L3 SQC registers 3

L3CNTLREG1 - L3 Control Register1

L3CNTLREG2 - L3 Control Register2

L3CNTLREG3 - L3 Control Register3
L3B2REG03 - L3 bank2 reg3 log error slice 0................................................................. 738
L3B2REG04 - L3 bank2 reg4 log error slice 0................................................................. 740
L3B2REG05 - L3 bank2 reg5 log error slice 0................................................................. 742
L3B2REG06 - L3 bank2 reg6 log error slice 0................................................................. 744
L3B2REG07 - L3 bank2 reg7 log error slice 0................................................................. 746
L3B3REG00 - L3 bank3 reg0 log error slice 0................................................................. 748
L3B3REG01 - L3 bank3 reg1 log error slice 0................................................................. 750
L3B3REG02 - L3 bank3 reg2 log error slice 0................................................................. 751
L3B3REG03 - L3 bank3 reg3 log error slice 0................................................................. 752
L3B3REG04 - L3 bank3 reg4 log error slice 0................................................................. 753
L3B3REG05 - L3 bank3 reg5 log error slice 0................................................................. 754
L3B3REG06 - L3 bank3 reg6 log error slice 0................................................................. 756
L3B3REG07 - L3 bank3 reg7 log error slice 0................................................................. 757
LPFCREG0 - First Buffer Size and Start........................................................................ 758
LPFCREG2 - Second Buffer Size .................................................................................. 759
LPFCREG03 - Error Reporting Reg Slice 0.................................................................. 760
LPFCREG04 - Frame count and Draw call number...................................................... 761
LPFCREG05 - SAVE Timer......................................................................................... 762
L3 Performance Counter Event Table.......................................................................... 763
LPFCREG06 - Event selection and base counters...................................................... 765
LPFCREG07 - Event Selection and Base Counters1................................................ 767
LPFCREG08 - MASTER start Timer........................................................................... 768
L3SYNC - L3 Cross Sync Control Register................................................................. 768
slmmsg - slm context save/restore msg................................................................. 769
SARBCSR - SARB config save msg.......................................................................... 769
SARERRST1 - SARB Error Status slice1..................................................................... 770
L3CDERRST11 - L3CD Error Status register 1 slice 1.............................................. 772
L3CDERRST12 - L3CD Error Status register 2 slice 1.............................................. 773
CLMREDS1 - Column Redundancy Slice 1................................................................. 774
LPCNTR1S1 - LPFC counter reg01 slice 1................................................................. 775
LPCNTR2S1 - LPFC counter reg02 slice 1................................................................. 775
LPCNTR3S1 - LPFC counter reg03 slice 1................................................................. 776
LPCNTR4S1 - LPFC counter reg04 slice 1................................................................. 776
LPCNTR5S1 - LPFC counter reg05 slice 1 .............................................................. 776
LPCNTR6S1 - LPFC counter reg06 slice 1 .............................................................. 776
LPCNTR7S1 - LPFC counter reg07 slice 1 .............................................................. 777
L3B0REG10 - L3 bank0 reg0 log error slice 1 ......................................................... 777
L3B0REG11 - L3 bank0 reg1 log error slice 1 ......................................................... 778
L3B0REG12 - L3 bank0 reg2 log error slice 1 ......................................................... 779
L3B0REG13 - L3 bank0 reg3 log error slice 1 ......................................................... 780
L3B0REG14 - L3 bank0 reg4 log error slice 1 ......................................................... 781
L3B0REG15 - L3 bank0 reg5 log error slice 1 ......................................................... 783
L3B0REG16 - L3 bank0 reg6 log error slice 1 ......................................................... 784
L3B0REG17 - L3 bank0 reg7 log error slice 1 ......................................................... 785
L3B1REG10 - L3 bank1 reg0 log error slice 1 ......................................................... 786
L3B1REG11 - L3 bank1 reg1 log error slice 1 ......................................................... 788
L3B1REG12 - L3 bank1 reg2 log error slice 1 ......................................................... 790
L3B1REG13 - L3 bank1 reg3 log error slice 1 ......................................................... 792
L3B1REG14 - L3 bank1 reg4 log error slice 1 ......................................................... 793
L3B1REG15 - L3 bank1 reg5 log error slice 1 ......................................................... 794
L3B1REG16 - L3 bank1 reg6 log error slice 1 ......................................................... 795
L3B1REG17 - L3 bank1 reg7 log error slice 1 ......................................................... 796
L3B2REG10 - L3 bank2 reg0 log error slice 1 ......................................................... 798
L3B2REG11 - L3 bank2 reg1 log error slice 1 ......................................................... 799
L3B2REG12 - L3 bank2 reg2 log error slice 1 ......................................................... 800
L3B2REG13 - L3 bank2 reg3 log error slice 1 ......................................................... 801
L3B2REG14 - L3 bank2 reg4 log error slice 1 ......................................................... 802
L3B2REG15 - L3 bank2 reg5 log error slice 1 ......................................................... 804
L3B2REG16 - L3 bank2 reg6 log error slice 1 ......................................................... 805
L3B2REG17 - L3 bank2 reg7 log error slice 1 ......................................................... 806
L3B3REG10 - L3 bank3 reg0 log error slice 1 ......................................................... 807
L3B3REG11 - L3 bank3 reg1 log error slice 1 ......................................................... 808
L3B3REG12 - L3 bank3 reg2 log error slice 1 ......................................................... 810
L3B3REG13 - L3 bank3 reg3 log error slice 1 ......................................................... 811
L3B3REG14 - L3 bank3 reg4 log error slice 1 ......................................................... 812
L3B3REG15 - L3 bank3 reg5 log error slice 1 ......................................................... 813
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Payload Messages</td>
<td>877</td>
</tr>
<tr>
<td>Generic Mode Root Thread</td>
<td>877</td>
</tr>
<tr>
<td>Root Thread from MEDIA_OBJECT_PRT</td>
<td>879</td>
</tr>
<tr>
<td>Root Thread from MEDIA_OBJECT_WALKER</td>
<td>881</td>
</tr>
<tr>
<td>Thread Spawn Message</td>
<td>881</td>
</tr>
<tr>
<td>Message Descriptor</td>
<td>883</td>
</tr>
<tr>
<td>Message Payload</td>
<td>883</td>
</tr>
<tr>
<td><strong>EU Overview</strong></td>
<td>886</td>
</tr>
<tr>
<td>Primary Usage Models</td>
<td>888</td>
</tr>
<tr>
<td>AOS and SOA Data Structures</td>
<td>889</td>
</tr>
<tr>
<td>SIMD4 Mode of Operation</td>
<td>891</td>
</tr>
<tr>
<td>SIMD4x2 Mode of Operation</td>
<td>892</td>
</tr>
<tr>
<td>SIMD16 Mode of Operation</td>
<td>894</td>
</tr>
<tr>
<td>SIMD8 Mode of Operation</td>
<td>896</td>
</tr>
<tr>
<td>Message Payload Containing a Header</td>
<td>897</td>
</tr>
<tr>
<td>Writebacks</td>
<td>898</td>
</tr>
<tr>
<td>Message Delivery Ordering Rules</td>
<td>899</td>
</tr>
<tr>
<td>Execution Mask and Messages</td>
<td>900</td>
</tr>
<tr>
<td>End-Of-Thread (EOT) Message</td>
<td>901</td>
</tr>
<tr>
<td>Message Description Syntax</td>
<td>902</td>
</tr>
<tr>
<td>Message Errors</td>
<td>903</td>
</tr>
<tr>
<td>Registers and Register Regions</td>
<td>905</td>
</tr>
<tr>
<td>Register Files</td>
<td>905</td>
</tr>
<tr>
<td>GRF Registers</td>
<td>906</td>
</tr>
<tr>
<td>ARF Registers</td>
<td>907</td>
</tr>
<tr>
<td>ARF Registers Overview</td>
<td>907</td>
</tr>
<tr>
<td>Access Granularity</td>
<td>908</td>
</tr>
<tr>
<td>Null Register</td>
<td>908</td>
</tr>
<tr>
<td>Address Register</td>
<td>909</td>
</tr>
<tr>
<td>Accumulator Registers</td>
<td>912</td>
</tr>
<tr>
<td>Flag Register</td>
<td>915</td>
</tr>
<tr>
<td>Channel Enable Register</td>
<td>916</td>
</tr>
<tr>
<td>SP Register</td>
<td>917</td>
</tr>
</tbody>
</table>
State Register .............................................................................................................................................. 918
Control Register.......................................................................................................................................... 921
Notification Registers ............................................................................................................................... 927
IP Register ..................................................................................................................................................... 929
TDR Registers .............................................................................................................................................. 929
Performance Registers............................................................................................................................. 932
Flow Control Registers.................................................................................................................................. 934
Immediate ......................................................................................................................................................... 935
Region Parameters ......................................................................................................................................... 936
Region Addressing Modes ........................................................................................................................... 941
Direct Register Addressing..................................................................................................................... 941
Register-Indirect Register Addressing with a 1x1 Index Region............................................. 942
Register-Indirect Register Addressing with a Vx1 Index Region ............................................ 943
Register-Indirect Register Addressing with a VxH Index Region ............................................ 944
Access Modes ................................................................................................................................................ 946
Execution Data Type....................................................................................................................................... 947
Register Region Restrictions ..................................................................................................................... 948
Destination Operand Description ............................................................................................................ 952
Destination Region Parameters............................................................................................................. 952
SIMD Execution Control ............................................................................................................................ 953
Predication ....................................................................................................................................................... 953
No Predication................................................................................................................................................. 955
Predication with Horizontal Combination ............................................................................................ 956
Predication with Vertical Combination .................................................................................................. 958
End of Thread ................................................................................................................................................ 959
Assigning Conditional Flags.................................................................................................................... 960
Destination Hazard....................................................................................................................................... 963
Non-present Operands ............................................................................................................................... 964
Instruction Prefetch ..................................................................................................................................... 965
ISA Introduction .......................................................................................................................................... 966
Introducing the Execution Unit .................................................................................................................. 967
EU Terms and Acronyms .......................................................................................................................... 970
Execution Units (EUs) ............................................................................................................................... 974
## Exception Descriptions

<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal Opcode</td>
<td>1004</td>
</tr>
<tr>
<td>Undefined Opcodes</td>
<td>1004</td>
</tr>
<tr>
<td>Software Exception</td>
<td>1004</td>
</tr>
<tr>
<td>Context Save and Restore</td>
<td>1004</td>
</tr>
<tr>
<td>Events That Do Not Generate Exceptions</td>
<td>1006</td>
</tr>
<tr>
<td>Illegal Instruction Format</td>
<td>1006</td>
</tr>
<tr>
<td>Malformed Message</td>
<td>1006</td>
</tr>
<tr>
<td>GRF Register Out of Bounds</td>
<td>1006</td>
</tr>
<tr>
<td>Hung Thread</td>
<td>1006</td>
</tr>
<tr>
<td>Instruction Fetch Out of Bounds</td>
<td>1006</td>
</tr>
<tr>
<td>FPU Math Errors</td>
<td>1007</td>
</tr>
<tr>
<td>Computational Overflow</td>
<td>1007</td>
</tr>
</tbody>
</table>

## Instruction Set Summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Set Characteristics</td>
<td>1008</td>
</tr>
<tr>
<td>SIMD Instructions and SIMD Width</td>
<td>1008</td>
</tr>
<tr>
<td>Instruction Operands and Register Regions</td>
<td>1008</td>
</tr>
<tr>
<td>Instruction Execution</td>
<td>1009</td>
</tr>
<tr>
<td>Instruction Machine Formats</td>
<td>1009</td>
</tr>
<tr>
<td>EU Instruction Formats</td>
<td>1012</td>
</tr>
<tr>
<td>Common Instruction Fields</td>
<td>1018</td>
</tr>
<tr>
<td>Instruction Operation Doubleword (DW0)</td>
<td>1024</td>
</tr>
<tr>
<td>Instruction Destination Doubleword (DW1)</td>
<td>1029</td>
</tr>
<tr>
<td>DW1 1-src and 2-src Instructions</td>
<td>1029</td>
</tr>
<tr>
<td>DW1 3-src Instructions</td>
<td>1033</td>
</tr>
<tr>
<td>Instruction Source 0 Doubleword 2 (DW2)</td>
<td>1035</td>
</tr>
<tr>
<td>DW2 1-src and 2-src Instructions</td>
<td>1035</td>
</tr>
<tr>
<td>Instruction Source 1 Doubleword 3 (DW3)</td>
<td>1040</td>
</tr>
<tr>
<td>EU Compact Instructions</td>
<td>1044</td>
</tr>
<tr>
<td>EU Compact Instruction Format</td>
<td>1045</td>
</tr>
<tr>
<td>EU Instruction Compaction Tables</td>
<td>1047</td>
</tr>
<tr>
<td>Opcode Encoding</td>
<td>1051</td>
</tr>
<tr>
<td>Move and Logic Instructions</td>
<td>1051</td>
</tr>
</tbody>
</table>
Render Engine Command Memory Interface

This chapter describes the memory-mapped registers associated with the Memory Interface, including brief descriptions of their use. The functions performed by some of these registers are discussed in more detail in the Memory Interface Functions, Memory Interface Instructions, and Programming Environment chapters.

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

This chapter describes the memory-mapped registers associated with the Memory Interface, including brief descriptions of their use. The functions performed by some of these registers are discussed in more detail in the Memory Interface Functions, Memory Interface Instructions, and Programming Environment chapters.

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

This section contains various registers for controls and modes.
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
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
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
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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>These registers should not be written with 0 for these projects. To avoid this, the MI_LOAD_REGISTER_MEMORY command which writes them from an address in memory which was written by a previous GPGPU_WALKER command will need to be checked with the following command sequence:</td>
<td></td>
</tr>
</tbody>
</table>

- MI_LOAD_REGISTER_MEMORY Xaddress, GPGPU_DISPATCHDIMX
- MI_CONDITIONAL_BATCH_BUFFER_END Xaddress, 0 // Compare X dimension to 0, end batch buffer if 0
- MI_LOAD_REGISTER_MEMORY GPGPU_DISPATCHDIMY
- MI_CONDITIONAL_BATCH_BUFFER_END Yaddress, 0 // Compare Y dimension to 0, end batch buffer if 0
- MI_LOAD_REGISTER_MEMORY GPGPU_DISPATCHDIMZ
- MI_CONDITIONAL_BATCH_BUFFER_END Zaddress, 0 // Compare Z dimension to 0, end batch buffer if 0
- GPGPU_WALKER // Walker with indirect dispatch

This way, if any dimension is 0 we would not execute the GPGPU_WALKER. This has the limitation that the indirect GPGPU_WALKER has to be the last WALKER of the batch buffer.

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

ALU PROGRAMMING

ALU 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.
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.

<table>
<thead>
<tr>
<th>ALU OPCODE</th>
<th>Operand-1</th>
<th>Operand-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bits</td>
<td>10 bits</td>
<td>10 bits</td>
</tr>
</tbody>
</table>

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 all '1' into SRCA or SRCB
- LOAD1 <SRCA, SRCB>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>SRCA/SRCB</td>
<td>R0,R1..R15</td>
</tr>
<tr>
<td>LOADINV</td>
<td>SRCA/SRCB</td>
<td>R0,R1..R15</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SUB</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AND</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OR</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>XOR</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>NOOP</td>
<td>N/A</td>
<td>NA</td>
</tr>
</tbody>
</table>
STORE Operation

Store instruction moves the content of the destination register (Operand1) into source register (Operand2). Destination register can be accumulator (ACCU), CF or ZF and destination register being GPR (R0, R1 ..R15). STORE has different flavors, where in one can load the inverted version of the source register into destination register. When CF or ZF are stored same value is replicated on all the 64bits.

//Loads ACCUMULATOR or Carry Flag or Zero Flag in to any of the mentioned generic registers Reg0 to 16 registers. In case of CF and ZF same value is replicated on all the 64 bits.

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

// Loads inverted (ACCMULATOR or Carry Flag or Zero Flag) in to any of the mentioned generic registers Reg0 to 15 registers

- STOREINV <R0.. R15>, <ACCU, CF, ZF> //Loads inverted (ACCMULATOR or Carry Flag or Zero Flag) in to any of the mentioned generic registers R0 to R15 registers

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Operand1</td>
<td>Operand2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>R0,R1..R15</td>
<td>ACCU/ZF/CF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOREINV</td>
<td>R0, R1.. R15</td>
<td>ACCU/ZF/CF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Operand1</td>
<td>Operand2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>SRCA/SRCB</td>
<td>REG0..REG15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOADINV</td>
<td>SRCA/SRCB</td>
<td>REG0..REG15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD0</td>
<td>SRCA/SRCB</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD1</td>
<td>SRCA/SRCB</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>REG0..REG15</td>
<td>ACCU/CF/ZF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOREINV</td>
<td>REG0..REG15</td>
<td>ACCU/CF/ZF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table for ALU OPCODE Encodings

<table>
<thead>
<tr>
<th>ALU OPCODE</th>
<th>OPCODE ENCODING</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>0x000</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x080</td>
</tr>
<tr>
<td>LOADINV</td>
<td>0x480</td>
</tr>
<tr>
<td>LOAD0</td>
<td>0x081</td>
</tr>
<tr>
<td>LOAD1</td>
<td>0x481</td>
</tr>
<tr>
<td>ADD</td>
<td>0x100</td>
</tr>
<tr>
<td>SUB</td>
<td>0x101</td>
</tr>
<tr>
<td>AND</td>
<td>0x102</td>
</tr>
<tr>
<td>OR</td>
<td>0x103</td>
</tr>
<tr>
<td>XOR</td>
<td>0x104</td>
</tr>
<tr>
<td>STORE</td>
<td>0x180</td>
</tr>
<tr>
<td>STOREINV</td>
<td>0x580</td>
</tr>
</tbody>
</table>

In the above mentioned table ALU Opcode Encodings look like some random numbers, rational behind those encodings is because of ALU Opcode is further broken down in to sub sections for ease of design implementation.

<table>
<thead>
<tr>
<th>PREFIX</th>
<th>OPCODE</th>
<th>SUBOPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PREFIX VALUE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Regular</td>
</tr>
<tr>
<td>1</td>
<td>Invert</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE VALUE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOOP</td>
</tr>
<tr>
<td>1</td>
<td>LOAD</td>
</tr>
<tr>
<td>2</td>
<td>ALU</td>
</tr>
<tr>
<td>3</td>
<td>STORE</td>
</tr>
<tr>
<td>ALU OPCODE</td>
<td>ENCODING</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>0x000</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x080</td>
</tr>
<tr>
<td>LOADINV</td>
<td>0x480</td>
</tr>
<tr>
<td>LOAD0</td>
<td>0x081</td>
</tr>
<tr>
<td>LOAD1</td>
<td>0x481</td>
</tr>
<tr>
<td>ADD</td>
<td>0x100</td>
</tr>
<tr>
<td>SUB</td>
<td>0x101</td>
</tr>
<tr>
<td>AND</td>
<td>0x102</td>
</tr>
<tr>
<td>OR</td>
<td>0x103</td>
</tr>
<tr>
<td>XOR</td>
<td>0x104</td>
</tr>
<tr>
<td>STORE</td>
<td>0x180</td>
</tr>
<tr>
<td>STOREINV</td>
<td>0x580</td>
</tr>
</tbody>
</table>

**Table for Register Encodings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Register Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0x0</td>
</tr>
<tr>
<td>R1</td>
<td>0x1</td>
</tr>
<tr>
<td>R2</td>
<td>0x2</td>
</tr>
<tr>
<td>R3</td>
<td>0x3</td>
</tr>
<tr>
<td>R4</td>
<td>0x4</td>
</tr>
<tr>
<td>R5</td>
<td>0x5</td>
</tr>
<tr>
<td>R6</td>
<td>0x6</td>
</tr>
<tr>
<td>R7</td>
<td>0x7</td>
</tr>
<tr>
<td>R8</td>
<td>0x8</td>
</tr>
<tr>
<td>R9</td>
<td>0x9</td>
</tr>
<tr>
<td>Register</td>
<td>Register Encoding</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>R10</td>
<td>0xa</td>
</tr>
<tr>
<td>R11</td>
<td>0xb</td>
</tr>
<tr>
<td>R12</td>
<td>0xc</td>
</tr>
<tr>
<td>R13</td>
<td>0xd</td>
</tr>
<tr>
<td>R14</td>
<td>0xe</td>
</tr>
<tr>
<td>R15</td>
<td>0xf</td>
</tr>
<tr>
<td>SRCA</td>
<td>0x20</td>
</tr>
<tr>
<td>SRCB</td>
<td>0x21</td>
</tr>
<tr>
<td>ACCU</td>
<td>0x31</td>
</tr>
<tr>
<td>ZF</td>
<td>0x32</td>
</tr>
<tr>
<td>CF</td>
<td>0x33</td>
</tr>
</tbody>
</table>

**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_PREDICATE
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 MItmp0, MItmp1, 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.
MI_SET_PREDICATE

Programming Note: Below is a table of command(s) that can be disabled by the MI_SET_PREDICATE command:

<table>
<thead>
<tr>
<th>Command</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DSTATE_URB_VS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_HS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_DS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_VS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_HS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_DS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_PS</td>
<td></td>
</tr>
<tr>
<td>MI_LOAD_REGISTER_IMM</td>
<td></td>
</tr>
<tr>
<td>MEDIA_VFE_STATE</td>
<td></td>
</tr>
<tr>
<td>MEDIA_OBJECT</td>
<td></td>
</tr>
<tr>
<td>MEDIA_OBJECT_WALKER</td>
<td></td>
</tr>
<tr>
<td>MEDIA_INTERFACE_DESCRIPTOR_LOAD</td>
<td></td>
</tr>
</tbody>
</table>

MI_SET_PREDICATE
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.)

**Programming Notes:**

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:

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
</table>

**Option 1:**
PIPE_CONTROL command with the CS Stall and the required write caches flushed with Post-Sync-Operation as Write Immediate Data followed by eight dummy MI_STORE_DATA_IMM (write to scratch space) commands.

Example:
- Workload-1
- PIPE_CONTROL (CS Stall, Post-Sync-Operation Write Immediate Data, Required Write Cache Flush bits set)
- MI_STORE_DATA_IMM (8 times) (Dummy data, Scratch Address)
- Workload-2 (Can use the data produce or output by Workload-1)

**Option 2:** This option has the overhead of TLBs getting invalidated.

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

Example:
- Workload-1 (3D/GPGPU/MEDIA)
- PIPE_CONTROL (TLB Invalidate, CS Stall, Post-Sync-Operation Write Immediate Data, Required Write Cache Flush bits set)
- WorkLoad-2 (Can use the data produce 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.

Note that 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. Postsynchronization 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 PSDEPTH 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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Hardware supports up to 16 pending PIPE_CONTROL flushes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Security</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td></td>
<td>PIPECONTROL with RO Cache Invalidation: Prior to programming a PIPECONTROL command with any of the RO cache invalidation bit set program a PIPECONTROL flush command with “CS stall” bit and “HDC Flush” bit set.</td>
</tr>
</tbody>
</table>

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

**Caches Invalidated/Flushed by PIPE_CONTROL Bit Settings**

<table>
<thead>
<tr>
<th>Write Cache Flush</th>
<th>Notification Enabled</th>
<th>Non-VF RO Cache Invalidate</th>
<th>VF RO Cache Invalidate</th>
<th>Marker Sent</th>
<th>Pipeline Marker Enable</th>
<th>Completion Requested</th>
<th>Top of Pipe Invalidate Pulse from CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**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 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Bits</th>
<th>Restriction</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRI Post Sync</td>
<td>23</td>
<td>Post Sync Operation ([15:14] of DW1) must be set to 0x0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Snapshot</td>
<td>19</td>
<td>Requires stall bit ([20] of DW1) set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count Reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Media</td>
<td>16</td>
<td>Requires stall bit ([20] of DW1) set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Clear</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect State</td>
<td>9</td>
<td>Requires stall bit ([20] of DW1) set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointers Disable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Data</td>
<td>21</td>
<td>Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sync GFDT</td>
<td>17</td>
<td>Post-Sync Operation ([15:14] of DW1) must be set to something other than '0' or 0x2520[13] must be set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLB inv</td>
<td>18</td>
<td>(All SKUs)(All Steppings): Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLB inv</td>
<td>18</td>
<td>Requires stall bit ([20] of DW1) set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post Sync Op</td>
<td>15:14</td>
<td>LRI Post Sync Operation ([23] of DW1) must be set to '0'.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Bit</th>
<th>Restrictions</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Stall</td>
<td>13</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Render Target Cache</td>
<td>12</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Flush</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth Cache Flush</td>
<td>0</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Stall Pixel Scoreboard</td>
<td>1</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Inst invalidate</td>
<td>11</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Tex invalidate</td>
<td>10</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>VF invalidate</td>
<td>4</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>Constant invalidate</td>
<td>3</td>
<td>No Restriction.</td>
<td></td>
</tr>
<tr>
<td>State Invalidate</td>
<td>2</td>
<td>No Restriction.</td>
<td></td>
</tr>
</tbody>
</table>
# Stall

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

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Bit</th>
<th>Restrictions</th>
<th>Project</th>
</tr>
</thead>
</table>
| Stall Bit | 20    | [All Stepping][All SKUs]:
|           |       | One of the following must also be set:
|           |       | • 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)               |
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

Entire context image consists of the Register/State Context, including the pipelined state section.
## RegisterState Context

### Context Color Codes Used

- **POWER CONTEXT**
- **RING CONTEXT**
- **MAIN CONTEXT**
- **EXTENDED CONTEXT**
- **URB_ATOMIC CONTEXT**

### Register Information

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Unit</th>
<th># of DW</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1083</td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RING_BUFFER_START</td>
<td>0x2038</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RING_BUFFER_CONTROL</td>
<td>0x203C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RVSYNC</td>
<td>0x2040</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RSYNC</td>
<td>0x2044</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RC_PSMI_CONTROL</td>
<td>0x2050</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RC_PWRCTX_MAXCNT</td>
<td>0x2054</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CTX_WA_PTR</td>
<td>0x2058</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOPID</td>
<td>0x2094</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HWSTAM</td>
<td>0x2098</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FF_THREAD_MODE</td>
<td>0x20A0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IMR</td>
<td>0x20A8</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EIR</td>
<td>0x20B0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EMR</td>
<td>0x20B4</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CMD_CCTL_0</td>
<td>0x20C4</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GAFS_Mode</td>
<td>0x212C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>UHPTR</td>
<td>0x2134</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BB_PREEMPT_ADDR</td>
<td>0x2148</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RING_BUFFER_HEAD_PREEMPT_REG</td>
<td>0x214C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CXT_SIZE</td>
<td>0x21A8</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CXT_OFFSET</td>
<td>0x21AC</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CXT_PIPESTATEBASE</td>
<td>0x21B0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PREEMPT_DLY</td>
<td>0x2214</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GFX_MODE</td>
<td>0x229C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MTCH_CID_RST</td>
<td>0x222C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT00L</td>
<td>0x2250</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT00H</td>
<td>0x2254</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>RLCONTENT01L</td>
<td>0x2258</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT01H</td>
<td>0x225C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT02L</td>
<td>0x2260</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT02H</td>
<td>0x2264</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT03L</td>
<td>0x2268</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT03H</td>
<td>0x226C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT10L</td>
<td>0x2270</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT10H</td>
<td>0x2274</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT11L</td>
<td>0x2278</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT11H</td>
<td>0x227C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT12L</td>
<td>0x2280</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT12H</td>
<td>0x2284</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT13L</td>
<td>0x2288</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLCONTENT13H</td>
<td>0x228C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SYNC_FLIP_STATUS</td>
<td>0x22D0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SYNC_FLIP_STATUS_1</td>
<td>0x22D4</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>OSBUFFER</td>
<td>0x23B0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>WAIT_FOR_RC6_EXIT</td>
<td>0x20CC</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SECOND_BB_ADDR</td>
<td>0x2114</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SECOND_BB_STATE</td>
<td>0x2118</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RVESYNC</td>
<td>0x2048</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SEMAPHORE-1/2</td>
<td>0x2680</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RS_OFFSET</td>
<td>0x21B4</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RS_PREEMPTION_HINT</td>
<td>0x24C0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CS_PREEMPTION_HINT</td>
<td>0x24BC</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CCID Register</td>
<td>0x2180</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SBB_PREEMPT_ADDRESS</td>
<td>0x213C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>URB_CTX_OFFSET</td>
<td>0x21B8</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_108D</td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GPM Data(Inc GAM)</td>
<td></td>
<td></td>
<td></td>
<td>142</td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1051</td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MBCunit</td>
<td></td>
<td>GPM</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1013</td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GCPunit</td>
<td></td>
<td>GPM</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_101F</td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GDTunit</td>
<td></td>
<td>GPM</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1047</td>
<td>GPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GAMunit</td>
<td></td>
<td>GPM</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>GPM</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1059</td>
<td>SPM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SPMunit</td>
<td></td>
<td>SPM</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SPM</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1015</td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Context Control</td>
<td>0x2244</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Ring Head Pointer Register</td>
<td>0x2034</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Ring Tail Pointer Register</td>
<td>0x2030</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Batch Buffer Current Head Register</td>
<td>0x2140</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Batch Buffer State Register</td>
<td>0x2110</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PPGTT Directory Cache Valid Register</td>
<td>0x2220</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PP_DIR_BASE</td>
<td>0x2228</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Read Offset in Pipelined State Page (8 CL aligned)</td>
<td>0x224C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Committed Vertex Number</td>
<td>0x21C4</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Committed Instance ID</td>
<td>0x21C8</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Committed Primitive ID</td>
<td>0x21CC</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>CS</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_10BF</td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>EXCC</td>
<td>0x2028</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_MODE</td>
<td>0x209C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>INSTMPT</td>
<td>0x20C0</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PR_CTR_CTL</td>
<td>0x2178</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PR_CTR_THRSH</td>
<td>0x217C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IA_VERTICES_COUNT</td>
<td>0x2310</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IA_PRIMITIVES_COUNT</td>
<td>0x2318</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>VS_INVOCATION_COUNT</td>
<td>0x2320</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>HS_INVOCATION_COUNT</td>
<td>0x2300</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DS_INVOCATION_COUNT</td>
<td>0x2308</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>GS_INVOCATION_COUNT</td>
<td>0x2328</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>GS_PRIMITIVES_COUNT</td>
<td>0x2330</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CL_INVOCATION_COUNT</td>
<td>0x2338</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>CL_PRIMITIVES_COUNT</td>
<td>0x2340</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PS_INVOCATION_COUNT_0</td>
<td>0x22C8</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PS_DEPTH_COUNT_0</td>
<td>0x22D8</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>VFSKPDP</td>
<td>0x2470</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP Register (LSB)</td>
<td>0x2358</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GPUGPU_DISPATCHDIMX</td>
<td>0x2500</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GPUGPU_DISPATCHDIMY</td>
<td>0x2504</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GPUGPU_DISPATCHDIMZ</td>
<td>0x2508</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_SRC0</td>
<td>0x2400</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_SRC0</td>
<td>0x2404</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_SRC1</td>
<td>0x2408</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_SRC1</td>
<td>0x240C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_DATA</td>
<td>0x2410</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_DATA</td>
<td>0x2414</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_RESULT</td>
<td>0x2418</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_END_OFFSET</td>
<td>0x2420</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_START_VERTEX</td>
<td>0x2430</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_VERTEX_COUNT</td>
<td>0x2434</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_INSTANCE_COUNT</td>
<td>0x2438</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_START_INSTANCE</td>
<td>0x243C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_BASE_VERTEX</td>
<td>0x2440</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GPGPU_THREADS_DISPATCHED</td>
<td>0x2290</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PS_INVOCATION_COUNT_1</td>
<td>0x22F0</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PS_DEPTH_COUNT_1</td>
<td>0x22F8</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>BB_START_ADDR</td>
<td>0x2150</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BB_ADD_DIFF</td>
<td>0x2154</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BB_OFFSET</td>
<td>0x2158</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RS_PREEMPT_STATUS</td>
<td>0x215C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CTX_SEMA_REG</td>
<td>0x24B4</td>
<td>CS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PRODUCE_COUNT_BTP</td>
<td>0x2480</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PRODUCE_COUNT_DX9_CONSTANTS</td>
<td>0x2484</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PRODUCE_COUNT_GATHER_CONSTANTS</td>
<td>0x248C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PARSED_COUNT_BTP</td>
<td>0x2490</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PARSED_COUNT_DX9_CONSTANTS</td>
<td>0x2494</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_PREDICATE_RESULT_1</td>
<td>0x241C</td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CS_GPR (1-16)</td>
<td>0x2600</td>
<td>CS</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>MI_TOPOLOGY_FILTER</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MI_URB_CLEAR</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MI_SET_APPID</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>PIPELINE_SELECT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATE_BASE_ADDRESS</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_VS</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_HS</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_DS</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_GS</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANTALLOC_PS</td>
<td></td>
<td>CS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POOL_ALLOC</td>
<td></td>
<td>CS</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_GATHER_POOL_ALLOC</td>
<td></td>
<td>CS</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC</td>
<td></td>
<td>CS</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MI_RS_CONTROL</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MI_URB_ATOMIC_ALLOC</td>
<td></td>
<td>CS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>CS</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SARB</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_10CF</td>
<td>SARB</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SARB Data</td>
<td></td>
<td>SARB</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SARB</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VS</td>
<td></td>
<td>SVG</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_VS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_VS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_VS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_VS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_HS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_HS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_HS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_HS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_HS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_TE</td>
<td></td>
<td>SVG</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DS</td>
<td></td>
<td>SVG</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_DS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_DS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_DS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_DS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_GS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_GS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_GS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_GS</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_GS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_STREAMOUT</td>
<td></td>
<td>SVG</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>3DSTATE_CLIP</td>
<td></td>
<td>SVG</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VIEWPORT_STATE_POINTERS_CL_SF</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SF</td>
<td></td>
<td>SVG</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SCISSOR_STATE_POINTERS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_MULTISAMPLE</td>
<td></td>
<td>SVG</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DRAWING_RECTANGLE</td>
<td></td>
<td>SVG</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SWTESS_BASE_ADDRESS</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SVG</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_WM</td>
<td></td>
<td>SVL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VIEWPORT_STATE_POINTERS_CC</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CC_STATE_POINTERS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DEPTH_STENCIL_STATE_POINTERS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLE_MASK</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SBE</td>
<td></td>
<td>SVL</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_PS</td>
<td></td>
<td>SVL</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PS</td>
<td></td>
<td>SVL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_PS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_PS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BLEND_STATE_POINTERS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1011</td>
<td>SVL</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Cache_Mode_0</td>
<td>0x7000</td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Cache_Mode_1</td>
<td>0x7004</td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GT_MODE</td>
<td>0x7008</td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FBC_RT_BASE_ADDR_REGISTER</td>
<td>0x7020</td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_MODE</td>
<td>0x7028</td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STATE_SIP</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DEPTH_BUFFER</td>
<td></td>
<td>SVL</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_STENCIL_BUFFER</td>
<td></td>
<td>SVL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_HIER_DEPTH_BUFFER</td>
<td></td>
<td>SVL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CLEAR_PARAMS</td>
<td></td>
<td>SVL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_RAST_MULTISAMPLE</td>
<td></td>
<td>SVL</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>GPGPU_CSR_BASE_ADDRESS</td>
<td></td>
<td>SVL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SVL</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>TDL</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1069</td>
<td>TDL</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TD_CTL</td>
<td>E400</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL2</td>
<td>E404</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_VF_VS_EMSK</td>
<td>E408</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_GS_EMSK</td>
<td>E40C</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>TD_WIZ_EMSK</td>
<td>E410</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_TS_EMSK</td>
<td>E428</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_HS_EMSK</td>
<td>E480</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_DS_EMSK</td>
<td>E484</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL</td>
<td>E500</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL2</td>
<td>E504</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_VF_VS_EMSK</td>
<td>E508</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_GS_EMSK</td>
<td>E50C</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_WIZ_EMSK</td>
<td>E510</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_TS_EMSK</td>
<td>E528</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_HS_EMSK</td>
<td>E580</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_DS_EMSK</td>
<td>E5B4</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL</td>
<td>E600</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL2</td>
<td>E604</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_VF_VS_EMSK</td>
<td>E608</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_GS_EMSK</td>
<td>E60C</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_WIZ_EMSK</td>
<td>E610</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_TS_EMSK</td>
<td>E628</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_HS_EMSK</td>
<td>E680</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_DS_EMSK</td>
<td>E6B4</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL</td>
<td>E700</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_CTL2</td>
<td>E704</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_VF_VS_EMSK</td>
<td>E708</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_GS_EMSK</td>
<td>E70C</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_WIZ_EMSK</td>
<td>E710</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_TS_EMSK</td>
<td>E728</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_HS_EMSK</td>
<td>E7B0</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TD_DS_EMSK</td>
<td>E7B4</td>
<td>TDL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>TDL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>WM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1003</td>
<td>WM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SuperSpan Count</td>
<td>0x5520</td>
<td>WM</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_POLY_STIPPLE_PATTERN</td>
<td>WM</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_AA_LINE_PARAMS</td>
<td>WM</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_POLY_STIPPLE_OFFSET</td>
<td>WM</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_LINE_STIPPLE</td>
<td>WM</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>WM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1005</td>
<td>SC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>3DSTATE_MONOFILTER_SIZE</td>
<td></td>
<td>SC</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CHROMA_KEY</td>
<td></td>
<td>SC</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SC</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_105D</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TDL DATA</td>
<td></td>
<td>VFE</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_105D</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TDL DATA</td>
<td></td>
<td>VFE</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_105D</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TDL DATA</td>
<td></td>
<td>VFE</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_105D</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TDL DATA</td>
<td></td>
<td>VFE</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_105D</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TDL DATA</td>
<td></td>
<td>VFE</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1029</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GW DATA</td>
<td></td>
<td>VFE</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1029</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GW DATA</td>
<td></td>
<td>VFE</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1029</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GW DATA</td>
<td></td>
<td>VFE</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1017</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TSG DATA</td>
<td></td>
<td>VFE</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1017</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TSG DATA</td>
<td></td>
<td>VFE</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1017</td>
<td>VFE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TSG DATA</td>
<td></td>
<td>VFE</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>---------------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>MEDIA_VFE_STATE</td>
<td></td>
<td>VFE</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MEDIA_CURBE_LOAD</td>
<td></td>
<td>VFE</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MEDIA_INTERFACE_DESCRIPTOR_LOAD</td>
<td></td>
<td>VFE</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MEDIA_OBJECT_PRT/GPGPU_WALKER</td>
<td></td>
<td>VFE</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>MEDIA_STATE_FLUSH</td>
<td></td>
<td>VFE</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>VFE</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_PALETTE_LOAD0</td>
<td></td>
<td>DM</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_PALETTE_LOAD1</td>
<td></td>
<td>DM</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>DM</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SOL</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load_Register_Immediate header</td>
<td>0x1100_1027</td>
<td>SOL</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SO_NUM_PRIMS_WRITTEN0</td>
<td>0x5200</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_NUM_PRIMS_WRITTEN1</td>
<td>0x5208</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_NUM_PRIMS_WRITTEN2</td>
<td>0x5210</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_NUM_PRIMS_WRITTEN3</td>
<td>0x5218</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_PRIM_STORAGE_NEEDED0</td>
<td>0x5240</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_PRIM_STORAGE_NEEDED1</td>
<td>0x5248</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_PRIM_STORAGE_NEEDED2</td>
<td>0x5250</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_PRIM_STORAGE_NEEDED3</td>
<td>0x5258</td>
<td>SOL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SO_WRITE_OFFSET0</td>
<td>0x5280</td>
<td>SOL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SO_WRITE_OFFSET1</td>
<td>0x5284</td>
<td>SOL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SO_WRITE_OFFSET2</td>
<td>0x5288</td>
<td>SOL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SO_WRITE_OFFSET3</td>
<td>0x528C</td>
<td>SOL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SO_BUFFER</td>
<td></td>
<td>SOL</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>SOL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SO_DECL_LIST</td>
<td></td>
<td>SOL</td>
<td>259</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_INDEX_BUFFER</td>
<td></td>
<td>VF</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VERTEX_BUFFERS</td>
<td></td>
<td>VF</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VERTEX_ELEMENTS</td>
<td></td>
<td>VF</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VF_STATISTICS</td>
<td></td>
<td>VF</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_VF</td>
<td></td>
<td>VF</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>RS</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POOL_ALLOC</td>
<td></td>
<td>RS</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_EDIT_VS</td>
<td></td>
<td>RS</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>RS</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_EDIT_GS</td>
<td></td>
<td>RS</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>RS</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_EDIT_HS</td>
<td></td>
<td>RS</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
<td>RS</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Unit</td>
<td># of DW</td>
<td>Security</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_EDIT_DS</td>
<td>RS</td>
<td>258</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_EDIT_PS</td>
<td>RS</td>
<td>258</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_GATHER_POOL_ALLOC</td>
<td>RS</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_END/NOOP ***</td>
<td>RS</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC</td>
<td>RS</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTF_VS(Global)</td>
<td>RS</td>
<td>1026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTI_VS(Global)</td>
<td>RS</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTB_VS(Global)</td>
<td>RS</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTF_VS(local)</td>
<td>RS</td>
<td>1026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTI_VS(local)</td>
<td>RS</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTB_VS(local)</td>
<td>RS</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_LOCAL_VALID_VS</td>
<td>RS</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTF_PS(Global)</td>
<td>RS</td>
<td>1026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTI_PS(Global)</td>
<td>RS</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTB_PS(Global)</td>
<td>RS</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTF_PS(local)</td>
<td>RS</td>
<td>1026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTI_PS(local)</td>
<td>RS</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_CONSTANTB_PS(local)</td>
<td>RS</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DSTATE_DX9_LOCAL_VALID_PS</td>
<td>RS</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_END</td>
<td>RS</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td>RS</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URB_ATOMIC_STORAGE</td>
<td>GAFS</td>
<td>8192</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
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.
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.
URB-Related State-Setting Commands

Several commands are used (among other things) to set state variables used in URB entry allocation --- specifically, the **Number of URB Entries** and the **URB Entry Allocation Size** state variables associated with various pipeline units. These state variables must be set-up prior to the issuing of a URB_FENCE command. (See the subsection on URB_FENCE.)

CS_URB_STATE (only) specifies these state variables for the common CS FF unit.

3DSTATE_PIPELINED_POINTERS sets the state variables for FF units in the 3D Pipeline, and MEDIA_STATE_POINTERS sets them for the Media pipeline. Depending on which pipeline is currently active, only one of these commands needs to be used. Note that these commands can also be reissued at a later time to change other state variables, though if a change is made to (a) any **Number of URB Entries** and the **URB Entry Allocation Size** state variables or (b) the **Maximum Number of Threads** state for the GS or CLIP FF units, a URB_FENCE command must follow.
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 assign that state must be issued before issuing 3DPRIMITIVE.
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** - Resource Streamer Preemption Status
- **MI_RS_CONTEXT**
- **MI_RS_CONTROL**
- **MI_RS_STORE_DATA_IMM**
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.

<table>
<thead>
<tr>
<th>Resource Streamer Sync Commands: Commands that RS Stops</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI_WAIT_FOR_EVENT</td>
<td></td>
</tr>
<tr>
<td>MI_RS_CONTROL</td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_END</td>
<td></td>
</tr>
<tr>
<td>MI_SEMAPHORE_MBOX</td>
<td></td>
</tr>
<tr>
<td>MI_SET_CONTEXT</td>
<td></td>
</tr>
<tr>
<td>MI_RS_CONTEXT</td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_START</td>
<td></td>
</tr>
<tr>
<td>MI_CONDITIONAL_BATCH_BUFFER_END</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 &quot;Binding Table Pool Enable&quot; 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:</td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_START (Resource Streamer Enabled)</td>
<td></td>
</tr>
<tr>
<td>PIPELINE_SELECT (3D)</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Enabled)</td>
<td></td>
</tr>
<tr>
<td>3D WORKLOAD MI_RS_CONTROL (Disable Resource Streamer)</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Disabled)</td>
<td></td>
</tr>
<tr>
<td>PIPELINE_SELECT (GPGPU)</td>
<td></td>
</tr>
<tr>
<td>GPGPU Workload</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POOL_ALLOC (Binding Table Pool Enabled)</td>
<td></td>
</tr>
<tr>
<td>MI_RS_CONTROL (Enable Resource Streamer)</td>
<td></td>
</tr>
<tr>
<td>3D WORKLOAD</td>
<td></td>
</tr>
<tr>
<td>MI_BATCH_BUFFER_END</td>
<td></td>
</tr>
</tbody>
</table>
# 3DSTATE_BINDING_TABLE_POOL_ALLOC

<table>
<thead>
<tr>
<th>Project</th>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>When switching between HW and SW binding table generation, SW must issue a state cache invalidate.</td>
<td></td>
</tr>
<tr>
<td>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 re-enabling the Binding Table Pool, re-enable the resource streamer thru the MI_RS_CONTROL command.</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** These are variable length commands: 3DSTATE_BINDING_TABLE_EDIT_HS, 3DSTATE_BINDING_TABLE_EDIT_DS, and 3DSTATE_BINDING_TABLE_EDIT_PS.
Gather Constants

In Dx10 the app can provide up to 16 constant buffers. 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. A new command, 3D_STATE_GATHER_CONSTANT_*, is added to offload the gather and packing functions from the driver.

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 fills the gather buffer when it executes a 3D_STATE_GATHER_CONSTANT_* command. Once the gather buffer has been filled, the command streamer executes the 3D_STATE_CONSTANT_* command 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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note: If the surface type is NULL, any fetch using the surface state base address is not bound by the size of the surface state and the fetch still occurs.</td>
<td></td>
</tr>
</tbody>
</table>

3DSTATE_GATHER_VS

Programming Note: The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note: The constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note: The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

Programming Note: When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. For example if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

Programming Note: If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. To avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero. Gather Pool must be disabled if executing a 3DSTATE_CONSTANT command unless the resource streamer is enabled to process a GATHER command (RS enabled batch and RS has not been disabled with a MI_RS_CONTROL command).
3DSTATE_GATHER_HS

**Programming Note:** The HW-generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Note:</strong> Furthermore, the constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. In other words if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. If the programmer wants to avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, it can send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero. Gather Pool must be disabled if executing a 3DSTATE_CONSTANT command unless the resource streamer is enabled to process a GATHER command(RS enabled batch and RS has not been disabled with a MI_RS_CONTROL command).

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Notes:</strong> The 3DSTATE_GATHER_* command is not committed to the resource streamer engine until the corresponding(same shader) 3DSTATE_BINDING_TABLE_POINTER_* command. For example, the 3DSTATE_GATHER_VS command does not actually generate a buffer in memory until the 3DSTATE_BINDING_TABLE_POINTERS_VS is parsed by the resource streamer.</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The following commands must be executed before any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid equal to zero:

- **3DPRIMITIVE** – To ensure resource streamer initiates produce prior to next command:
  - Indirect Parameter Enable = 0
  - UAV Coherency Required = 0
  - Predicate Enable = 0
  - End Offset Enable = 0
  - Vertex Access Type = SEQUENTIAL
  - Primitive Topology Type = 3DPRIM_POINTLIST
  - Vertex Count Per Instance = 0
  - Start Vertex Location = 0
  - Instance Count = 0
  - Start Instance Location = 0
  - Base Vertex Location = 0

- **MI_RS_STORE_DATA_IMM** – To force engine idle prior to executing next instruction. Write must
occur to address that does not corrupt memory:

<table>
<thead>
<tr>
<th></th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Streamer Flush</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note:** The following commands must be executed *before* any `3DSTATE_GATHER_CONSTANT_*` command that has **Constant Buffer Valid** greater than zero:

- **3DPRIMITIVE** – To ensure resource streamer initiates produce prior to next command:
  
  Indirect Parameter Enable = 0  
  UAV Coherency Required = 0  
  Predicate Enable = 0  
  End Offset Enable = 0  
  Vertex Access Type = SEQUENTIAL  
  Primitive Topology Type = 3DPRIM_POINTLIST  
  Vertex Count Per Instance = 0  
  Start Vertex Location = 0  
  Instance Count = 0  
  Start Instance Location = 0  
  Base Vertex Location = 0

**Note:** The following commands must be executed *following* any `3DSTATE_GATHER_CONSTANT_*` command that has **Constant Buffer Valid** greater than zero:

- **MI_RS_STORE_DATA_IMM** – To force engine idle prior to executing next instruction. Write must occur to address that does not corrupt memory:
  
  Resource Streamer Flush = 1

- **MI_RS_STORE_DATA_IMM** – To force all previous writes to coherent memory point. Write must occur to address that does not corrupt memory:
  
  Resource Streamer Flush = 1

- **3DSTATE_GATHER_CONSTANT_PS** – To ensure correct timing of sync between resource streamer and render pipeline:
  
  Constant Buffer Valid = 0

- **3DSTATE_CONSTANT_PS**:
  
  Constant Buffer 1 Read Length = 0  
  Constant Buffer 0 Read Length = 0  
  Constant Buffer 3 Read Length = 0  
  Constant Buffer 2 Read Length = 0
3DSTATE_GATHER_DS

**Programming Note:** The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Note:</strong> The constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* command must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. For example if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. To avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero. Gather Pool must be disabled if executing a 3DSTATE_CONSTANT command unless the resource streamer is enabled to process a GATHER command (RS enabled batch and RS has not been disabled with a MI_RS_CONTROL command).

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Note:</strong> The 3DSTATE_GATHER_* command is not committed to the resource streamer engine until the corresponding (same shader) 3DSTATE_BINDING_TABLE_POINTER_* command. For example, the 3DSTATE_GATHER_VS command will not actually generate a buffer in memory until the 3DSTATE_BINDING_TABLE_POINTERS_VS is parsed by the resource streamer.</td>
<td></td>
</tr>
</tbody>
</table>

3DSTATE_GATHER_GS

**Programming Note:** The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furthermore, the constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bit in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* command must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. In other words if a 3DSTATE_GATHER_CONSTANT_VS occurs
before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. If the programmer wants to avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, it can send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero. Gather Pool must be disabled if executing a 3DSTATE_CONSTANT command unless the resource streamer is enabled to process a GATHER command (RS enabled batch and RS has not been disabled with a MI_RS_CONTROL command).

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
</table>

**Programming Note:** The 3DSTATE_GATHER_* command is not committed to the resource streamer engine until the corresponding (same shader) 3DSTATE_BINDING_TABLE_POINTER_* command. For example, the 3DSTATE_GATHER_VS command will not actually generate a buffer in memory till the 3DSTATE_BINDING_TABLE_POINTERS_VS is parsed by the resource streamer.

**Note:** The following commands must be executed prior to any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid equal to zero:

- **3DPRIMITIVE** – To ensure resource streamer initiates produce prior to next command:
  - Indirect Parameter Enable = 0
  - UAV Coherency Required = 0
  - Predicate Enable = 0
  - End Offset Enable = 0
  - Vertex Access Type = SEQUENTIAL
  - Primitive Topology Type = 3DPRIM_POINTLIST
  - Vertex Count Per Instance = 0
  - Start Vertex Location = 0
  - Instance Count = 0
  - Start Instance Location = 0
  - Base Vertex Location = 0

- **MI_RS_STORE_DATA_IMM** – To force engine idle prior to executing next instruction. Write must occur to address that will not corrupt memory.
  - Resource Streamer Flush = 1

**Note:** The following commands must be executed prior to any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid greater than zero:

- **3DPRIMITIVE** – To ensure resource streamer initiates produce prior to next command:
  - Indirect Parameter Enable = 0
  - UAV Coherency Required = 0
  - Predicate Enable = 0
  - End Offset Enable = 0
  - Vertex Access Type = SEQUENTIAL
  - Primitive Topology Type = 3DPRIM_POINTLIST
  - Vertex Count Per Instance = 0
  - Start Vertex Location = 0
Instance Count            = 0
Start Instance Location   = 0
Base Vertex Location      = 0

Note: The following commands must be executed **following** any 3DSTATE_GATHER_CONSTANT_* command that has **Constant Buffer Valid** greater than zero:

- **MI_RS_STORE_DATA_IMM** – To force engine idle prior to executing next instruction. Write must occur to address that will not corrupt memory:
  - Resource Streamer Flush = 1

- **MI_RS_STORE_DATA_IMM** – To force all previous writes to coherent memory point. Write must occur to address that will not corrupt memory:
  - Resource Streamer Flush = 1

- **3DSTATE_GATHER_CONSTANT_PS** – To ensure correct timing of sync between resource streamer and render pipeline:
  - Constant Buffer Valid = 0

- **3DSTATE_CONSTANT_PS**:
  - Constant Buffer 1 Read Length = 0
  - Constant Buffer 0 Read Length = 0
  - Constant Buffer 3 Read Length = 0
  - Constant Buffer 2 Read Length = 0
3DSTATE_GATHER_PS

Programming Note: The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
</table>

Programming Note: The constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.

Programming Note: The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

Programming Note: When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. For example if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

Programming Note: If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. To avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero. Gather Pool must be disabled if executing a 3DSTATE_CONSTANT command unless the resource streamer is enabled to process a GATHER command (RS enabled batch and RS has not been disabled with a MI_RS_CONTROL command).
**Programming Note:** The gather constant feature has a simple all or nothing model. If the gather constants are enable, the driver must enable the gather pool and use 3D_STATE_GATHER_CONSTANT_* cmds to gather and load the URB. If the gather buffer is disabled the driver must use the existing 3D_STATE_CONSTANT_* cmds to load the URB.

**Programming Note:** The gather constants can only be enabled if the binding table generator is also enabled.

**3DSTATE_GATHER_POOL_ALLOC**
3DSTATE_GATHER_CONSTANT_VS

**Programming Note:** The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Note:</strong></td>
<td>The constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
</tr>
</tbody>
</table>

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. For example if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. To avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero.

3DSTATE_GATHER_CONSTANT_VS
Programming Note: The HW-generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note: Furthermore, the constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note: The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

Programming Note: When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore, the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. In other words if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

Programming Note: If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. If the programmer wants to avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, it can send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Notes: The 3DSTATE_GATHER_* command is not committed to the resource streamer engine until the corresponding(same shader) 3DSTATE_BINDING_TABLE_POINTER_* command. For example, the 3DSTATE_GATHER_VS command does not actually generate a buffer in memory until the 3DSTATE_BINDING_TABLE_POINTERS_VS is parsed by the resource streamer.</td>
<td></td>
</tr>
</tbody>
</table>

[HSW] Note: The following commands must be executed before any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid equal to zero:

- **3DPRIMITIVE** - To ensure resource streamer initiates produce prior to next command:
  - Indirect Parameter Enable = 0
  - UAV Coherency Required = 0
  - Predicate Enable = 0
  - End Offset Enable = 0
  - Vertex Access Type = SEQUENTIAL
  - Primitive Topology Type = 3DPRIM_POINTLIST
  - Vertex Count Per Instance = 0
  - Start Vertex Location = 0
  - Instance Count = 0
  - Start Instance Location = 0
  - Base Vertex Location = 0

- **MI_RS_STORE_DATA_IMM** - To force engine idle prior to executing next instruction. Write must occur to address that does not corrupt memory:
  - Resource Streamer Flush = 1
Note: The following commands must be executed *before* any 3DSTATE_GATHER_CONSTANT_* command that has **Constant Buffer Valid** greater than zero:

- **3DPRIMITIVE** - To ensure resource streamer initiates produce prior to next command:
  
  - Indirect Parameter Enable = 0
  - UAV Coherency Required = 0
  - Predicate Enable = 0
  - End Offset Enable = 0
  - Vertex Access Type = SEQUENTIAL
  - Primitive Topology Type = 3DPRIM_POINTLIST
  - Vertex Count Per Instance = 0
  - Start Vertex Location = 0
  - Instance Count = 0
  - Start Instance Location = 0
  - Base Vertex Location = 0

Note: The following commands must be executed *following* any 3DSTATE_GATHER_CONSTANT_* command that has **Constant Buffer Valid** greater than zero:

- **MI_RS_STORE_DATA_IMM** - To force engine idle prior to executing next instruction. Write must occur to address that does not corrupt memory:
  
  - Resource Streamer Flush = 1

- **MI_RS_STORE_DATA_IMM** - To force all previous writes to coherent memory point. Write must occur to address that does not corrupt memory:
  
  - Resource Streamer Flush = 1

- **3DSTATE_GATHER_CONSTANT_PS** - To ensure correct timing of sync between resource streamer and render pipeline:
  
  - Constant Buffer Valid = 0

- **3DSTATE_CONSTANT_PS**:
  
  - Constant Buffer 1 Read Length = 0
  - Constant Buffer 0 Read Length = 0
  - Constant Buffer 3 Read Length = 0
  - Constant Buffer 2 Read Length = 0

---

**3DSTATE_GATHER_CONSTANT_HS**
3DSTATE_GATHER_DS

3DSTATE_GATHER_CONSTANT_DS
**3DSTATE_GATHER_CONSTANT_GS**

**Programming Note:** The HW generated binding table must be enabled to use this command.

Furthermore, the constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWORDs, but the gather table entries are 16 bit in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. In other words if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. If the programmer wants to avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, it can send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero.

**Programming Note:** The 3DSTATE_GATHER_* command is not committed to the resource streamer engine until the corresponding (same shader) 3DSTATE_BINDING_TABLE_POINTER_* command. For example, the 3DSTATE_GATHER_VS command will not actually generate a buffer in memory till the 3DSTATE_BINDING_TABLE_POINTERS_VS is parsed by the resource streamer.

**Note:** The following commands must be executed **prior to** any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid equal to zero:

- **3DPRIMITIVE** - To ensure resource streamer initiates produce prior to next command:
  
<table>
<thead>
<tr>
<th></th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect Parameter Enable</td>
<td>0</td>
</tr>
<tr>
<td>UAV Coherency Required</td>
<td>0</td>
</tr>
<tr>
<td>Predicate Enable</td>
<td>0</td>
</tr>
<tr>
<td>End Offset Enable</td>
<td>0</td>
</tr>
<tr>
<td>Vertex Access Type</td>
<td>SEQUENTIAL</td>
</tr>
<tr>
<td>Primitive Topology Type</td>
<td>3DPRIM_POINTLIST</td>
</tr>
<tr>
<td>Vertex Count Per Instance</td>
<td>0</td>
</tr>
<tr>
<td>Start Vertex Location</td>
<td>0</td>
</tr>
<tr>
<td>Instance Count</td>
<td>0</td>
</tr>
<tr>
<td>Start Instance Location</td>
<td>0</td>
</tr>
<tr>
<td>Base Vertex Location</td>
<td>0</td>
</tr>
</tbody>
</table>

- **MI_RS_STORE_DATA_IMM** - To force engine idle prior to executing next instruction. Write must occur to address that will not corrupt memory.
  
  | Resource Streamer Flush        | 1   |

**Note:** The following commands must be executed **prior to** any 3DSTATE_GATHER_CONSTANT_* command that has Constant Buffer Valid equal to zero:
**Constant Buffer Valid** greater than zero:

- **3DPRIMITIVE** - To ensure resource streamer initiates produce prior to next command:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect Parameter Enable</td>
<td>0</td>
</tr>
<tr>
<td>UAV Coherency Required</td>
<td>0</td>
</tr>
<tr>
<td>Predicate Enable</td>
<td>0</td>
</tr>
<tr>
<td>End Offset Enable</td>
<td>0</td>
</tr>
<tr>
<td>Vertex Access Type</td>
<td>SEQUENTIAL</td>
</tr>
<tr>
<td>Primitive Topology Type</td>
<td>3DPRIM_POINTLIST</td>
</tr>
<tr>
<td>Vertex Count Per Instance</td>
<td>0</td>
</tr>
<tr>
<td>Start Vertex Location</td>
<td>0</td>
</tr>
<tr>
<td>Instance Count</td>
<td>0</td>
</tr>
<tr>
<td>Start Instance Location</td>
<td>0</td>
</tr>
<tr>
<td>Base Vertex Location</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:** The following commands must be executed **following** any 3DSTATE_GATHER_CONSTANT_* command that has **Constant Buffer Valid** greater than zero:

- **MI_RS_STORE_DATA_IMM** - To force engine idle prior to executing next instruction. Write must occur to address that will not corrupt memory:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Streamer Flush</td>
<td>1</td>
</tr>
</tbody>
</table>

- **MI_RS_STORE_DATA_IMM** - To force all previous writes to coherent memory point. Write must occur to address that will not corrupt memory:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Streamer Flush</td>
<td>1</td>
</tr>
</tbody>
</table>

- **3DSTATE_GATHER_CONSTANT_PS** - To ensure correct timing of sync between resource streamer and render pipeline:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Buffer Valid</td>
<td>0</td>
</tr>
</tbody>
</table>

- **3DSTATE_CONSTANT_PS**:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Buffer 1 Read Length</td>
<td>0</td>
</tr>
<tr>
<td>Constant Buffer 0 Read Length</td>
<td>0</td>
</tr>
<tr>
<td>Constant Buffer 3 Read Length</td>
<td>0</td>
</tr>
<tr>
<td>Constant Buffer 2 Read Length</td>
<td>0</td>
</tr>
</tbody>
</table>

**3DSTATE_GATHER_CONSTANT_GS**
3DSTATE_GATHER_CONSTANT_PS

**Programming Note:** The HW generated binding table must be enabled to use this command.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Note:</strong> The constant buffer block (group of aligned 16 binding table entries) must be set before this command is issued.</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** The length of the gather table is derived from the total length of the command. The command length is in DWords, but the gather table entries are 16 bits in length. If there is an unused odd entry at the end of the command the channel mask should be set to all 0s.

**Programming Note:** When a 3DSTATE_GATHER_CONSTANT_* command is used there must be a matching 3DSTATE_CONSTANT_* command. Furthermore the 3DSTATE_CONSTANT_* must occur in the same order as the 3DSTATE_GATHER_CONSTANT_* command. For example if a 3DSTATE_GATHER_CONSTANT_VS occurs before a 3DSTATE_GATHER_CONSTANT_PS, then the 3DSTATE_CONSTANT_VS must occur before the 3DSTATE_CONSTANT_PS.

**Programming Note:** If Gather pool is enabled, there must be a corresponding 3DSTATE_GATHER_CONSTANT command with any 3DSTATE_CONSTANT for any particular shader. To avoid any update to the Gather pool, and yet program the 3DSTATE_CONSTANT for a particular shader, send a 3DSTATE_GATHER_CONSTANT command with all valid bits set to zero.

3DSTATE_GATHER_CONSTANT_PS
Dx9 Constant Buffer Generation

The Dx9 constant model is a set of registers 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*_* commands 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 2nd 4KB unused.

**Programming Note:** The Dx9 constant buffer feature has a simple all or nothing model.

**Programming Note:** A maximum of 16,383 Binding Tables are allowed in any batch buffer.

**Programming Note:** 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

**Programming Note:** The 3DSTATE_DX9_CONSTANTB_VS is a variable length command.

3DSTATE_DX9_CONSTANTB_VS
3DSTATE_DX9_LOCAL_VALID_VS

<table>
<thead>
<tr>
<th>Offset</th>
<th>Cache Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>256 4-component Float Constants</td>
</tr>
<tr>
<td>0x0fff</td>
<td>16 4-component integer constants</td>
</tr>
<tr>
<td>0x1000</td>
<td>16 1-component boolean constants</td>
</tr>
<tr>
<td>0x1050</td>
<td>unused</td>
</tr>
<tr>
<td>0x1fff</td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Cache Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
</tr>
<tr>
<td>0x0fff</td>
<td>63</td>
</tr>
<tr>
<td>0x1000</td>
<td>64</td>
</tr>
<tr>
<td>0x103f</td>
<td></td>
</tr>
<tr>
<td>0x1040</td>
<td>68</td>
</tr>
<tr>
<td>0x104f</td>
<td></td>
</tr>
<tr>
<td>0x1050</td>
<td>unused</td>
</tr>
<tr>
<td>0x1fff</td>
<td></td>
</tr>
</tbody>
</table>

**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 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 in order to complete the sample instruction.

<table>
<thead>
<tr>
<th>Subfunction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture Coordinate</td>
<td>Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include: cube map intersection.</td>
</tr>
<tr>
<td>Processing</td>
<td></td>
</tr>
<tr>
<td>Texel Address Generation</td>
<td>The Sampling Engine will determine 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 mipmap weighting factors to be used in the subsequent filtering operations.</td>
</tr>
<tr>
<td>Texel Fetch</td>
<td>The required texel samples will be read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.</td>
</tr>
<tr>
<td>Texture Palette Lookup</td>
<td>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.</td>
</tr>
<tr>
<td>Shadow Pre-Filter Compare</td>
<td>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.</td>
</tr>
<tr>
<td>Texel Filtering</td>
<td>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.</td>
</tr>
<tr>
<td>Texel Color Gamma Linearization</td>
<td>Performs optional gamma decorrection on texel RGB (not A) values.</td>
</tr>
<tr>
<td>Subfunction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Denoise/Deinterlacer</td>
<td>Performs denoise and deinterlacing functions for video content</td>
</tr>
<tr>
<td>8x8 Video Scaler</td>
<td>Performs scaling using an 8x8 filter</td>
</tr>
</tbody>
</table>
**Sampling Engine**

The 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 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 in order to complete the sample instruction.

<table>
<thead>
<tr>
<th>Subfunction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture Coordinate Processing</td>
<td>Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include: cube map intersection.</td>
</tr>
<tr>
<td>Texel Address Generation</td>
<td>The Sampling Engine will determine 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 mplevel weighting factors to be used in the subsequent filtering operations.</td>
</tr>
<tr>
<td>Texel Fetch</td>
<td>The required texel samples will be read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.</td>
</tr>
<tr>
<td>Texture Palette Lookup</td>
<td>For streams which have &quot;paletted&quot; texture surface formats, this function uses the &quot;index&quot; values read from the texture map to look up texel color data from the texture palette.</td>
</tr>
<tr>
<td>Shadow Pre-Filter Compare</td>
<td>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.</td>
</tr>
<tr>
<td>Texel Filtering</td>
<td>Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This &quot;combination&quot; ranges from simply passing through a &quot;nearest&quot; 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.</td>
</tr>
<tr>
<td>Texel Color Gamma Linearization</td>
<td>Performs optional gamma decorrection on texel RGB (not A) values.</td>
</tr>
<tr>
<td>Denoise/Deinterlacer</td>
<td>Performs denoise and deinterlacing functions for video content</td>
</tr>
<tr>
<td>8x8 Video Scaler</td>
<td>Performs scaling using an 8x8 filter</td>
</tr>
</tbody>
</table>
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

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 as part of the pixel shader kernel in the Gen4 Core.

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

Note:
Face origin is here

- I face
- J face
+ I face
+ J face

I
J

abs(I0) > abs(J0)
Selects +I face

I0,J0

B.6878.01
Texel Address Generation

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

The Texel Address Generation function of the Sampling Engine is responsible for determining how the texture maps are to be sampled. Outputs of this function include the number of texel samples to be taken, 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 mipmap maps are provided and enabled, and an object pixel is found to cover multiple texels (e.g., when a textured object is located a significant distance from the viewer), the device will sample the mipmap level(s) offering a texel/pixel ratio as close to 1.0 as possible.

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

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

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:

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

**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) $\text{MinLOD}$ and $\text{MaxLOD}$ state variables prior to use in Min/Mag Determination.

$\text{MaxLOD}$ 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 $\text{MaxLOD}$ must be stored in memory, or operation is UNDEFINED.

$\text{MinLOD}$ specifies the highest resolution mip level (minimum LOD value) that can be accessed, where LOD=$\text{-}$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.

$\text{MinLOD}$ and $\text{MaxLOD}$ 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 $\text{MinLOD}$ is 4.5 and $\text{MipFilter}$ 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 $\text{BaseMipLevel}$ state variable is subtracted from the biased and clamped LOD. The $\text{BaseMipLevel}$ state variable therefore has the effect of selecting the "base" mip level used to compute Min/Map Determination. (This was added to match OpenGL semantics). Setting $\text{BaseMipLevel}$ to 0 has the effect of using the highest-resolution mip level as the base map.

If the biased and clamped LOD is non-positive, the texture is being magnified, and a single (high-resolution) miplevel will be sampled and filtered using the $\text{MagFilter}$ state variable. At this point the computed LOD is reset to 0.0. Note that LOD Clamping can restrict access to high-resolution miplevels.

If the biased LOD is positive, the texture is being minified. In this case the $\text{MipFilter}$ 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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias:</td>
<td>S4.8</td>
</tr>
<tr>
<td>MinLod:</td>
<td>U4.8</td>
</tr>
<tr>
<td>MaxLod:</td>
<td>U4.8</td>
</tr>
<tr>
<td>Base:</td>
<td>U4.1</td>
</tr>
<tr>
<td>MIPCnt:</td>
<td>U4</td>
</tr>
<tr>
<td>SurfMinLod:</td>
<td>U4.8</td>
</tr>
<tr>
<td>ResMinLod:</td>
<td>U4.8</td>
</tr>
</tbody>
</table>

PerSampleMinLOD: float32

\[
\begin{align*}
\text{MinLod} &= \max(\text{MinLod}, \text{PerSampleMinLOD}) \\
\text{AdjMaxLod} &= \min(\text{MaxLod}, \text{MIPCnt}) \\
\text{AdjMinLod} &= \min(\text{MinLod}, \text{MIPCnt}) \\
\text{AdjPR_minLOD} &= \text{ResMinLod} - \text{SurfMinLod} \\
\text{AdjMinLod} &= \max(\text{AdjMinLod}, \text{AdjPR_minLOD}) \\
\text{Out_of_Bounds} &= \text{AdjPR_minLOD} > \text{MIPCnt} \\
\end{align*}
\]

if ( sample_b )
   LOD += Bias + bias_parameter
else if ( sample_l or ld )
   LOD = Bias + lod_parameter
else
   LOD += Bias

<table>
<thead>
<tr>
<th>Project</th>
<th>Pseudocode</th>
</tr>
</thead>
</table>
| HSW     | PreClamp = LODPreClampEnable 
if ( PreClamp )
   LOD = min(LOD, MaxLod) 
   LOD = max(LOD, MinLod) |

MagMode = (LOD - Base <= 0)

<table>
<thead>
<tr>
<th>Project</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>MagClampMipNone = 1</td>
</tr>
<tr>
<td>HSW</td>
<td>MagClampMipNone = LODClampMagnificationMode == MAG_CLAMP_MIPNONE</td>
</tr>
</tbody>
</table>

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

**Project**

<table>
<thead>
<tr>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
</tr>
<tr>
<td>LOD = min(LOD, ceil(AdjMaxLod))</td>
</tr>
<tr>
<td>LOD = max(LOD, floor(AdjMinLod))</td>
</tr>
<tr>
<td>LOD += 0.5</td>
</tr>
<tr>
<td>LOD = floor(LOD)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>// MipFlt = Linear</td>
</tr>
<tr>
<td>LOD = min(LOD, AdjMaxLod)</td>
</tr>
<tr>
<td>LOD = max(LOD, AdjMinLod)</td>
</tr>
<tr>
<td>TriBeta = frac(LOD)</td>
</tr>
<tr>
<td>LOD0 = floor(LOD)</td>
</tr>
<tr>
<td>LOD1 = LOD0 + 1</td>
</tr>
<tr>
<td>if ( ! lod ) // &quot;LOD&quot; message type</td>
</tr>
<tr>
<td>Lod += SurfMinLod</td>
</tr>
</tbody>
</table>

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.

### 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:

<table>
<thead>
<tr>
<th>MipFilter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPFILTER_NONE</td>
<td>Mipmapping is DISABLED. Apply a single filter on the highest resolution map available (after LOD clamping).</td>
</tr>
<tr>
<td>MIPFILTER_NEAREST</td>
<td>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 mipmap level. LOD Clamping may further restrict this mipmap selection.</td>
</tr>
<tr>
<td>MIPFILTER_LINEAR</td>
<td>Apply a filter on the two closest mipmap 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 mipmap levels (and the blend factor between them).</td>
</tr>
</tbody>
</table>

When minifying and MIPFILTER_NEAREST is selected, the computed LOD is rounded to the nearest mipmap 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 mipmap level selected by the truncated LOD, and the next higher (lower resolution) mipmap 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.
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.

<table>
<thead>
<tr>
<th>Sampler[]Min/MagFilter value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAPFILTER_NEAREST</td>
<td>Supported on all surface types. The texel nearest to the pixel's U,V,Q coordinate is read and output from the filter.</td>
</tr>
<tr>
<td>MAPFILTER_LINEAR</td>
<td>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.</td>
</tr>
<tr>
<td>MAPFILTER_ANISOTROPIC</td>
<td>Not supported on buffer or 3D surfaces. A projection of the pixel onto the texture map is generated and &quot;subpixel&quot; 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.</td>
</tr>
<tr>
<td>MAPFILTER_MONO</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

**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 four texels surrounding the pixel center are chosen for the bilinear filter. 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.

**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 \((u, v)\) relative to the pixel center at \((u, v)\) is given by the following:

\[
\begin{align*}
\mathbf{u}_f &= u - \frac{N_u}{2} \\
\mathbf{v}_f &= v - \frac{N_v}{2}
\end{align*}
\]
\[S = \frac{1}{N_u \times N_v}\]

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

**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.

<table>
<thead>
<tr>
<th><strong>TC([X,Y,Z]) Control</strong></th>
<th><strong>Operation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXCOORDMODE_CLAMP</td>
<td>Clamp to the texel value at the edge of the map.</td>
</tr>
<tr>
<td>TEXCOORDMODE_CLAMP_BORDER</td>
<td>Use the texture map’s border color for any texel samples falling outside the map. The border color is specified via a pointer in SAMPLER_STATE.</td>
</tr>
<tr>
<td>TEXCOORDMODE_HALF_BORDER</td>
<td>Similar to CLAMP_BORDER except texels outside of the map are clamped to a value halfway between the edge texel and the border color.</td>
</tr>
<tr>
<td>TEXCOORDMODE_WRAP</td>
<td>Upon crossing an edge of the map, repeat at the other side of the map in the same dimension.</td>
</tr>
<tr>
<td>TEXCOORDMODE_CUBE</td>
<td>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.</td>
</tr>
<tr>
<td>TEXCOORDMODE_MIRROR</td>
<td>Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.</td>
</tr>
<tr>
<td>TEXCOORDMODE_MIRROR_ONCE</td>
<td>Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.</td>
</tr>
</tbody>
</table>

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.

**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).
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.

Texture Wrap vs. Mirror Addressing Mode

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).

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.
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.

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.

<table>
<thead>
<tr>
<th><code>ShadowFunction</code></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFILTEROP_ALWAYS</td>
<td>0.0</td>
</tr>
<tr>
<td>PREFILTEROP_NEVER</td>
<td>1.0</td>
</tr>
<tr>
<td>PREFILTEROP_LESS</td>
<td>(texel &lt; ref) 0.0 : 1.0</td>
</tr>
<tr>
<td>PREFILTEROP_EQUAL</td>
<td>(texel == ref) 0.0 : 1.0</td>
</tr>
<tr>
<td>PREFILTEROP_LEQUAL</td>
<td>(texel &lt;= ref) 0.0 : 1.0</td>
</tr>
<tr>
<td>PREFILTEROP_GREATER</td>
<td>(texel &gt; ref) 0.0 : 1.0</td>
</tr>
<tr>
<td>PREFILTEROP_NOTEQUAL</td>
<td>(texel != ref) 0.0 : 1.0</td>
</tr>
<tr>
<td>PREFILTEROP_GEQUAL</td>
<td>(texel &gt;= ref) 0.0 : 1.0</td>
</tr>
</tbody>
</table>

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.

**Programming Note:** Refer to the Surface Formats table in section `RENDER_SURFACE_STATE` for the specific surface formats that are supported with shadow mapping.
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 \textit{MinFilter}, \textit{MagFilter}, and \textit{MipFilter} are used to control the filtering of texel values. The \textit{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 \textit{MinFilter} and \textit{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 from gamma=2.4 space to gamma=1.0 space by applying a $(1/2.4) = ^0.4167$ 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 a 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

SURFACE_STATE for Deinterlace, sample_8x8, and VME

This section contains media surface state definitions.

MEDIA_SURFACE_STATE

Restrictions: The Faulting modes described in the MEMORY_OBJECT_CONTROL_STATE should be set to the same for the multi-surface Video Analytics functions like "LBP Correlation" and "Correlation Search" for both the surfaces.

SAMPLER_STATE

SAMPLER_STATE has different formats, depending on the message type used. The sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

### Project:

| Min LOD and Max LOD | fields need range increased from [0.0,13.0] to [0.0,14.0] and fractional bits increased from 6 to 8. This requires a few fields to be moved as indicated in the text. |

#### SAMPLER_STATE

#### SAMPLER_STATE for Sample_8x8 Message

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEF Filter Type</td>
<td>was dropped and is assumed to be Detailed filter.</td>
</tr>
<tr>
<td>IEF Filter Size</td>
<td>was dropped and assumed to be 5x5.</td>
</tr>
<tr>
<td>IEF Bypass</td>
<td>– If we have Y/G-channel masked then the IEF bypass should always be forced to 1.</td>
</tr>
</tbody>
</table>

#### DEINTERLACE_SAMPLER_STATE

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.
Restrictions

1. VDIWalker can be enabled only when frame is aligned to block size of 16x4 if DI is enabled (interlaced) and 16x8 if DN only (Progressive).

2. When VDIWalker Frame Sharing is enabled driver should dispatch same number of Media Objects to both half slice by explicitly programming half slice destination select as 01b and 10b alternately. (Note: Dispatch of threads should be in ping pong fashion to have load balance between both Halfslice and better L3 utilization.)

3. For VDIWalker disabled mode (when frame size is not aligned to 16x4 or 16x8) it is recommended to have a simplified SW walker. Using Half Slice Destination Select 00b will affect performance significantly.

Dispatch of Media Object Commands for VDIWalker Enabled

1. Frame Sharing is Disabled:
   a. Program all MO commands to have Half Slice destination select as either 01b or 10b.
   b. Y_stride programmed in Sampler State will be ignored.

2. Frame Sharing Enabled:
   a. If Frame_height (in blocks) % 2 = 0 (where block height = 4 when DI enabled, 8 when DN only) dispatch MO in ping pong fashion.
   b. Y_Stride of 0,1,2,3 is valid and VDIwalker will divide frame into multiple slices based on stride value.
   c. If Frame_height (in blocks) % 2 > 0, then dispatch MO in ping pong fashion and all threads for blocks from residual row to be sent to Half Slice0.

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;
reminder_media_obj_cnt_HS1 = fw * Y_Blocks_Remainder_HS1;
reminder_media_obj_cnt_HS2 = fw * Y_Blocks_Remainder_HS2;

ping_pong_media_obj_cnt = total_media_obj_cnt – (reminder_media_obj_cnt_HS1 +
reminder_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 < reminder_media_obj_cnt_HS1; i++ ) {
    dispatch_media_object_hs1;
}

for ( i = 0; i < reminder_media_obj_cnt_HS2; i++ ) {
    dispatch_media_object_hs2;
}

SAMPLER_8x8_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.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:24</td>
<td>Border Color Alpha</td>
<td>UNORM8</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>Border Color Blue</td>
<td>UNORM8</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>Border Color Green</td>
<td>UNORM8</td>
</tr>
<tr>
<td>0</td>
<td>7:0</td>
<td>Border Color Red</td>
<td>UNORM8</td>
</tr>
<tr>
<td>1</td>
<td>31:0</td>
<td>Border Color Red</td>
<td>IEEE_FP</td>
</tr>
<tr>
<td>2</td>
<td>31:0</td>
<td>Border Color Green</td>
<td>IEEE_FP</td>
</tr>
<tr>
<td>3</td>
<td>31:0</td>
<td>Border Color Blue</td>
<td>IEEE_FP</td>
</tr>
<tr>
<td>4</td>
<td>31:0</td>
<td>Border Color Alpha</td>
<td>IEEE_FP</td>
</tr>
<tr>
<td>5</td>
<td>31:16</td>
<td>Border Color Green</td>
<td>FLOAT16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td>Border Color Red</td>
<td>FLOAT16</td>
</tr>
<tr>
<td>6</td>
<td>31:16</td>
<td>Border Color Alpha</td>
<td>FLOAT16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td>Border Color Blue</td>
<td>FLOAT16</td>
</tr>
<tr>
<td>7</td>
<td>31:16</td>
<td>Border Color Green</td>
<td>FLOAT16</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Format</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UNORM16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Border Color Red</strong></td>
<td>UNORM16</td>
</tr>
<tr>
<td>8</td>
<td>31:16</td>
<td><strong>Border Color Alpha</strong></td>
<td>UNORM16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Border Color Blue</strong></td>
<td>UNORM16</td>
</tr>
<tr>
<td>9</td>
<td>31:16</td>
<td><strong>Border Color Green</strong></td>
<td>SNORM16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Border Color Red</strong></td>
<td>SNORM16</td>
</tr>
<tr>
<td>10</td>
<td>31:16</td>
<td><strong>Border Color Alpha</strong></td>
<td>SNORM16</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Border Color Blue</strong></td>
<td>SNORM16</td>
</tr>
<tr>
<td>11</td>
<td>31:24</td>
<td><strong>Border Color Alpha</strong></td>
<td>SNORM8</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td><strong>Border Color Blue</strong></td>
<td>SNORM8</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td><strong>Border Color Green</strong></td>
<td>SNORM8</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>Border Color Red</strong></td>
<td>SNORM8</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Integer Surface Format – Different Types</th>
<th>Surface formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bpc, 128 bpt case (4 types)</td>
<td>R32G32B32A32_UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_SINT</td>
</tr>
<tr>
<td>16 bpc, 64 bpt case (5 types)</td>
<td>R16G16B16A16_UINT, R10G10B10A2_UINT</td>
</tr>
<tr>
<td></td>
<td>X32_TYPELESS_G8X24_UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_SINT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_SINT</td>
</tr>
<tr>
<td>32 bpc, 64 bpt case (2 types)</td>
<td>R32G32_UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32_SINT</td>
</tr>
<tr>
<td>8 bpc, 32 bpt cases (9 types)</td>
<td>R8G8B8A8(UINT)</td>
</tr>
<tr>
<td></td>
<td>R8G8_UINT</td>
</tr>
<tr>
<td></td>
<td>R8_UINT</td>
</tr>
<tr>
<td></td>
<td>X24_TYPELESS_G8_UINT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8_UINT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_SINT</td>
</tr>
<tr>
<td></td>
<td>R8G8_SINT</td>
</tr>
<tr>
<td></td>
<td>R8_SINT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8_SINT</td>
</tr>
<tr>
<td>16 bpc, 32 bpt cases (4 types)</td>
<td>R16G16_UINT</td>
</tr>
<tr>
<td></td>
<td>R16_UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16_SINT</td>
</tr>
<tr>
<td></td>
<td>R16_SINT</td>
</tr>
<tr>
<td>32 bpc, 32 bpt case (2 types)</td>
<td>R32_UINT</td>
</tr>
<tr>
<td></td>
<td>R32_SINT</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case 1: R32G32B32A32_UINT</strong> (32 bpc, 128 bpt, 4 channels)</td>
<td></td>
</tr>
</tbody>
</table>
| DWORDN   | 31:0 | **Border Color Red** ui32 (integer unclamp) Format: INT32  
|         |     | **Texture Border Color Mode** = DX10/OGL |
| DWORDN+1 | 31:0 | **Border Color Green** ui32 (integer unclamp) Format: INT32  
|         |     | **Texture Border Color Mode** = DX10/OGL |
| DWORDN+2 | 31:0 | **Border Color Blue** ui32 (integer unclamp) Format: INT32  
|         |     | **Texture Border Color Mode** = DX10/OGL |
| DWORDN+3 | 31:0 | **Border Color Alpha** ui32 (integer unclamp) Format: INT32  
|         |     | **Texture Border Color Mode** = DX10/OGL |

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

Each of the values in the above table would have be to programmed as sint32 value.
Case 3: R32G32B32_UINT (32 bpc, 128 bpt, 3 channel)
R/G/B values would be programmed like in Case1. Alpha channel value at DWORDN+3 would have to be programmed as Integer 1.

Case 4: R32_UINT (32 bpc, 32 bpt, 1 channel)

<table>
<thead>
<tr>
<th>DWORDN</th>
<th>31:0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Border Color Red ui32 (integer unclamp)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td>DWORDN+1</td>
<td>31:0</td>
<td><strong>Border Color Green ui32 (integer unclamp)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should be programmed as integer 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td>DWORDN+2</td>
<td>31:0</td>
<td><strong>Border Color Blue ui32 (integer unclamp)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should be programmed as integer 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td>DWORDN+3</td>
<td>31:0</td>
<td><strong>Border Color Alpha ui32 (integer unclamp)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should be programmed as integer 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

Case 5: R16G16B16A16_UINT (16 bpc, 64 bpt, 4 channel, UINT)

<table>
<thead>
<tr>
<th>DWORDN</th>
<th>15:0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Border Color Red clamp to uint16</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td><strong>Border Color Green clamp to uint16</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td>DWORDN+1</td>
<td>31:0</td>
<td><strong>Reserved</strong></td>
</tr>
</tbody>
</table>
### Case 6: R8G8B8A8_SINT (8 bpc, 32 bpt, 4 channels, SINT)

| DWORDN+2 | 15:0 | Border Color Blue clamp to uint16  
| Format: U16  
| Texture Border Color Mode = DX10/OGL |
| 31:16 | Border Color Alpha clamp to uint16  
| Format: U16  
| Texture Border Color Mode = DX10/OGL |
| DWORDN+3 | 31:0 | Reserved  
| Format: MBZ |

| DWORDN | 7:0 | Border Color Red clamp to sint8  
| Format: U8  
| Texture Border Color Mode = DX10/OGL |
| 15:8 | Border Color Green clamp to sint8  
| Format: U8  
| Texture Border Color Mode = DX10/OGL |
| 23:16 | Border Color Blue clamp to sint8  
| Format: U8  
| Texture Border Color Mode = DX10/OGL |
| 31:24 | Border Color Alpha clamp to sint8  
| Format: U8  
| Texture Border Color Mode = DX10/OGL |
| DWORDN+1 | 31:0 | Reserved  
| Format: MBZ |
| DWORDN+2 | 31:0 | Reserved  
| Format: MBZ |
## Case 7: R32G32_UINT (32bpc, 64bpt, 2 channel case)

| DWORDN+3 | 31:0 | Reserved
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: MBZ</td>
</tr>
</tbody>
</table>

**Texture Border Color Mode** = DX10/OGL

<table>
<thead>
<tr>
<th>DWORDN</th>
<th>31:0</th>
<th>Border Color Red ui32 (integer unclamp)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

| DWORDN+1 | 31:0 | Reserved
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: MBZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DWORDN+2</th>
<th>31:0</th>
<th>Border Color Green ui32 (integer unclamp)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: INT32</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

| DWORDN+3 | 31:0 | Reserved
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: MBZ</td>
</tr>
</tbody>
</table>

## Case 8: R8_UINT (8 bpc, 32 bpt, 1 channel case)

<table>
<thead>
<tr>
<th>DWORDN</th>
<th>7:0</th>
<th>Border Color Red clamp to uint8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>15:8</th>
<th>Border Color Green</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Should be programmed as integer 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>23:16</th>
<th>Border Color Blue clamp to uint8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Should be programmed as integer 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
</tbody>
</table>

<p>|          | 31:24 | Border Color Alpha clamp to uint8 |</p>
<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should be programmed as integer 1. Format: U8</td>
<td><strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td><strong>DWORDN+1</strong> 31:0</td>
<td><strong>Reserved</strong> Format: MBZ</td>
</tr>
<tr>
<td><strong>DWORDN+2</strong> 31:0</td>
<td><strong>Reserved</strong> Format: MBZ</td>
</tr>
<tr>
<td><strong>DWORDN+3</strong> 31:0</td>
<td><strong>Reserved</strong> Format: MBZ</td>
</tr>
</tbody>
</table>

**Case 9: R16G16_UINT (16 bpc, 32 bpt case)**

<table>
<thead>
<tr>
<th>DWORDN</th>
<th>15:0</th>
<th><strong>Border Color Red clamp to uint16</strong> Format: U16 <strong>Texture Border Color Mode</strong> = DX10/OGL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31:16</td>
<td><strong>Border Color Green clamp to uint16</strong> Format: U16 <strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td><strong>DWORDN+1</strong> 31:0</td>
<td><strong>Reserved</strong> Format: MBZ</td>
<td></td>
</tr>
<tr>
<td><strong>DWORDN+2</strong> 15:0</td>
<td><strong>Border Color Blue clamp to uint16</strong> Format: U16 <strong>Texture Border Color Mode</strong> = DX10/OGL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td><strong>Border Color Alpha clamp to uint16</strong> Program as Integer 1. Format: U16 <strong>Texture Border Color Mode</strong> = DX10/OGL</td>
</tr>
<tr>
<td><strong>DWORDN+3</strong> 31:0</td>
<td><strong>Reserved</strong></td>
<td></td>
</tr>
<tr>
<td>Project:</td>
<td>HSW</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format: MBZ</td>
<td></td>
</tr>
</tbody>
</table>

3DSTATE_CHROMA_KEY
3DSTATE_SAMPLER_PALETTE_LOAD0
3DSTATE_MONOFILTER_SIZE
**Messages**

**Restrictions:**

- Use of any message to the Sampling Engine function with the **End of Thread** bit set in the message descriptor is not allowed.

**Initiating Message**

**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 lower 8 bits of the execution mask forms 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 lower 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.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Header Present: Specifies whether the message includes a header phase. If the header is not present (this field is zero), all of the fields normally contained in the header are assumed to be 0. Format = Enable</td>
</tr>
<tr>
<td>18:17</td>
<td>SIMD Mode: Specifies the SIMD mode of the message being sent. Format = U2 0 = SIMD4x2 1 = SIMD8 2 = SIMD16 3 = SIMD32/64</td>
</tr>
<tr>
<td>16:12</td>
<td>Message Type: Specifies the type of message being sent. Format = U5 Refer to the table in the Payload Parameter Definition section for encoding details.</td>
</tr>
</tbody>
</table>
| 11:8  | Sampler Index: Specifies the index into the sampler state table. Ignored for "ld", "resinfo", "sampleinfo" and "cache_flush" type messages. Format = U4 Range = [0,15] Programming Notes:  
- For the deinterlace message, this field must be a multiple of 2 (even).  
- For the sample_8x8 message, this field must be a multiple of 4. |
| 7:0   | Binding Table Index: Specifies the index into the binding table. Ignored for "cache_flush" type messages. Format = U8 Range = [0,255] |


**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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer</strong>: Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for &quot;ld&quot; and &quot;resinfo&quot; message types. This pointer is relative to the Dynamic State Base Address. Format = DynamicStateOffset[31:5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M0.2 spans so many rows, many for various projects, that the DWord value is repeated in each row.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.2</td>
<td>31:22</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>21</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>20</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>19:18</td>
<td><strong>SIMD32/64 Output Format Control</strong> 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 Downsamplerd 2: 8 bit Full 3: 8 bit Chrominance Downsampled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This field is ignored for sample_8x8 messages if the Function is not AVS and
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MinMaxFilter. For MinMaxFilter only 16bit Full and 8bit Full modes are supported. This field is ignored and not used for HDC write message. 0: 16 bit Full 1: 16 bit Chrominance Downsampling 2: 8 bit Full 3: 8 bit Chrominance Downsampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>17:16</td>
<td><strong>Gather4 Source Channel Select:</strong> 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  <strong>Programming Note:</strong> For gather4*_c messages, this field must be set to 0 (Red channel).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>14</td>
<td><strong>Blue Write Channel Mask:</strong> See Alpha Write Channel Mask.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>13</td>
<td><strong>Green Write Channel Mask:</strong> See Alpha Write Channel Mask.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>12</td>
<td><strong>Red Write Channel Mask:</strong> See Alpha Write Channel Mask.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>11:8</td>
<td><strong>U Offset:</strong> The u offset from the _aoffimmi modifier on the &quot;sample&quot; or &quot;ld&quot; 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.  <strong>Programming Notes:</strong>  - This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.  - This field is ignored if the &quot;offu&quot; parameter is included in the gather4* messages.  - <strong>Note:</strong> HSW offu/offv are calculated in normalized space and hence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Payload Parameter Definition

The following sections show all of the messages supported by the sampling engine. The message type field in the message descriptor in combination with the message length determines 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 instruction column indicates the DX10 shader instructions expected to be translated to each message type.

All parameters are of type IEEE_Float, except those in the ld and resinfo instruction message types, which are of type S31. Any parameter indicated with a blank entry in the table is unused. A message register containing only unused parameters is not included as part of the message. The response lengths given
below assume that all channels are unmasked. SIMD16 messages with masked channels have reduced response lengths.

For the SIMD32/SIMD64 messages, the input message is not defined in terms of parameters. "H" is 1 if the header is present, 0 otherwise.

**SIMD32/SIMD64 Messages**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Mnemonic</th>
<th>Payload Layout</th>
<th>Message Length</th>
<th>Response Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>sample_unorm</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>8 **</td>
</tr>
<tr>
<td>00010</td>
<td>sample_unorm+killpix</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>9 **</td>
</tr>
<tr>
<td>00011</td>
<td>sample_8x8</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>16 *</td>
</tr>
<tr>
<td>01000</td>
<td>deinterlace</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>†</td>
</tr>
<tr>
<td>01100</td>
<td>sample_unorm</td>
<td>Media</td>
<td>H + 1</td>
<td>8 **</td>
</tr>
<tr>
<td>01010</td>
<td>sample_unorm+killpix</td>
<td>Media</td>
<td>H + 1</td>
<td>9 **</td>
</tr>
<tr>
<td>01011</td>
<td>sample_8x8</td>
<td>Media</td>
<td>H + 1</td>
<td>16 *</td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Parameter 0 is required except for the sampleinfo message, which has no parameter 0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD Mode</th>
<th>Message Length</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD4x2</td>
<td>H + (N/4)</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>H + N</td>
<td></td>
</tr>
<tr>
<td>SIMD8D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMD16</td>
<td>H + (2*N)</td>
<td></td>
</tr>
</tbody>
</table>

The response lengths are computed as follows:

**Determining Response Lengths**

<table>
<thead>
<tr>
<th>Project</th>
<th>SIMD Mode</th>
<th>Message Type</th>
<th>Return Format</th>
<th>Response Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD4x2</td>
<td>All</td>
<td>32-bit</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>sample+killpix</td>
<td>32-bit</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>All other message types</td>
<td>32-bit</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SIMD16</td>
<td>All</td>
<td>32-bit</td>
<td>8 *</td>
<td></td>
</tr>
<tr>
<td>SIMD16</td>
<td>All</td>
<td>16-bit</td>
<td>4 *</td>
<td></td>
</tr>
</tbody>
</table>
Notes for Determining Response Lengths Table

<table>
<thead>
<tr>
<th>Project</th>
<th>Symbol</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Project:**

**SIMD4x2 Messages**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Mnemonic</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>sample_l</td>
<td>u v r ai lod</td>
</tr>
<tr>
<td>00100</td>
<td>sample_d</td>
<td>u v r ai dudx dudy dvdx dvdy drdx drdy</td>
</tr>
<tr>
<td>00110</td>
<td>sample_l_c</td>
<td>u v r ai ref lod</td>
</tr>
<tr>
<td>00111</td>
<td>ld</td>
<td>u v r lod</td>
</tr>
<tr>
<td>01000</td>
<td>gather4</td>
<td>u v r ai</td>
</tr>
<tr>
<td>01010</td>
<td>resinfo</td>
<td>lod</td>
</tr>
<tr>
<td>01011</td>
<td>sampleinfo</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>gather4_c</td>
<td>u v r ai ref</td>
</tr>
<tr>
<td>10001</td>
<td>gather4_po</td>
<td>u v r ai offu offv</td>
</tr>
<tr>
<td>10010</td>
<td>gather4_po_c</td>
<td>u v r ref offu offv</td>
</tr>
<tr>
<td>10100</td>
<td>sample_d_c</td>
<td>u v r ai dudx dudy dvdx dvdy drdx drdy ref</td>
</tr>
<tr>
<td>11100</td>
<td>ld2dms_w</td>
<td>u v r si mcsl mcsh</td>
</tr>
<tr>
<td>11101</td>
<td>ld_mcs</td>
<td>u v r</td>
</tr>
<tr>
<td>11110</td>
<td>ld2dms</td>
<td>u v r si mcs</td>
</tr>
</tbody>
</table>

**SIMD32/SIMD64 Messages**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Mnemonic</th>
<th>Payload Layout</th>
<th>Message Length</th>
<th>Response Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>sample_unorm</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>8 **</td>
</tr>
<tr>
<td>00010</td>
<td>sample_unorm+killpix</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>9 **</td>
</tr>
<tr>
<td>01000</td>
<td>deinterlace</td>
<td>Pixel Shader</td>
<td>H + 1</td>
<td>†</td>
</tr>
</tbody>
</table>
### Message Types

**Programming Note:** 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.

The behavior of each message type is as follows:

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Project</th>
<th>Description or Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels. One, two, or three parameters may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. 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 is not supported in SIMD4x2 mode.</td>
</tr>
<tr>
<td>sample</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample</td>
<td></td>
<td>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td>
</tr>
<tr>
<td>sample+killpix</td>
<td></td>
<td>The surface is sampled as in the sample message type. 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. The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.</td>
</tr>
<tr>
<td>Message Type</td>
<td>Project</td>
<td>Description or Restriction</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>SURFTYPE_CUBE.</td>
<td>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.</td>
</tr>
<tr>
<td>sample+killpix</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_b</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels, then the value in the parameter is added to the LOD for each pixel. 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. The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. The Surface Format of the associated surface cannot be MONO8. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. sample_b is not supported in SIMD4x2 mode.</td>
</tr>
<tr>
<td>sample_b</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_l</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. LOD is not computed, but instead is taken from the lod parameter.</td>
</tr>
<tr>
<td>sample_lz</td>
<td></td>
<td>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td>
</tr>
<tr>
<td>sample_l</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_lz</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_c</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. All four coordinates must be specified; however v and r may not be used depending on the indicated surface type. The ai parameter indicates the array index for a cube surface. The ref parameter specifies the reference value that is compared against the red channel of the sampled surface, and the texel is replaced with either white or black depending on the result of the comparison. The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. The Surface Format of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. With sample_c, MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed even for surface formats that are listed as not supporting filtering in the surface formats table. Use of the SIMD4x2 form of sample_c without Force LOD to Zero enabled in the message header.</td>
</tr>
<tr>
<td>sample_c</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. All four coordinates must be specified; however v and r may not be used depending on the indicated surface type. The ai parameter indicates the array index for a cube surface. The ref parameter specifies the reference value that is compared against the red channel of the sampled surface, and the texel is replaced with either white or black depending on the result of the comparison. The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. The Surface Format of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. With sample_c, MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed even for surface formats that are listed as not supporting filtering in the surface formats table. Use of the SIMD4x2 form of sample_c without Force LOD to Zero enabled in the message header.</td>
</tr>
</tbody>
</table>

152
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Project</th>
<th>Description or Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_c</td>
<td></td>
<td>is not allowed, as it is not possible for the hardware to compute LOD for SIMD4x2 messages. <strong>sample_c</strong> is not supported in SIMD4x2 mode. Use of <strong>sample_c</strong> with DX9 <strong>Texture Border Color Mode</strong> and either of the following is undefined: Use of <strong>sample_c</strong> with SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, and A32_FLOAT. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. 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.</td>
</tr>
<tr>
<td>sample_c_lz</td>
<td></td>
<td>The WGF <strong>sample_c_lz</strong> instruction is implemented by issuing the <strong>sample_c</strong> message with Force LOD to Zero enabled in the message header or by issuing the <strong>sample_l_c</strong> message with the LOD parameter set to zero.</td>
</tr>
<tr>
<td>sample_b_c</td>
<td></td>
<td>This is a combination of <strong>sample_b</strong> and <strong>sample_c</strong>. Both the LOD bias and reference values are delivered. All restrictions applying to both <strong>sample_b</strong> and <strong>sample_c</strong> must be honored.</td>
</tr>
<tr>
<td>sample_l_c</td>
<td></td>
<td>This is a combination of <strong>sample_l</strong> and <strong>sample_c</strong>. Both the LOD and reference values are delivered. All restrictions applying to both <strong>sample_l</strong> and <strong>sample_c</strong> must be honored. However, unlike <strong>sample_c</strong>, <strong>sample_l_c</strong> is allowed as a SIMD4x2 message.</td>
</tr>
<tr>
<td>sample_g</td>
<td></td>
<td>The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The r coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The r coordinate contains the faceid, and the r gradients are ignored by hardware. 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.</td>
</tr>
<tr>
<td>sample_d</td>
<td></td>
<td>The Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_g</td>
<td></td>
<td>If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.</td>
</tr>
<tr>
<td>sample_d</td>
<td></td>
<td><strong>Note:</strong> The hardware will not set dv* and dq* parameters to zero if they are not provided in the message. Thus these parameters must be provided even for surfaces that do not use them, 1D and 2D surfaces.</td>
</tr>
<tr>
<td>sample_g_c</td>
<td></td>
<td>This is a combination of <strong>sample_g</strong> and <strong>sample_c</strong>. Both the gradients for calculating LOD and reference values are delivered. All restrictions applying to both <strong>sample_g</strong> and <strong>sample_c</strong> must be honored. However, unlike <strong>sample_c</strong>, <strong>sample_g_c</strong> is allowed as a SIMD4x2 message.</td>
</tr>
<tr>
<td>resinfo</td>
<td></td>
<td>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. The width, height, and depth may be shifted right, per pixel, by the LOD value provided in the lod parameter to give the dimensions of the specified mip level. The lod parameter is an unsigned 32-bit integer in this mode (note that sending a signed 32-bit integer always has the same effect, as negative values are out-of-range when interpreted as unsigned integers). The</td>
</tr>
</tbody>
</table>
Sampler State Pointer and Sampler Index are ignored.
For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, if the delivered LOD is outside of the range [0..MipCount-1], the returned values in the red, green, and blue channels are 0s.

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>Project</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURFTYPE_1D</td>
<td>(Width+1) &gt;&gt; LOD</td>
<td>Surface Array Depth+1 : 0</td>
<td>0</td>
<td>MIPCount</td>
<td></td>
</tr>
<tr>
<td>SURFTYPE_2D</td>
<td>(Width+1) &gt;&gt; LOD</td>
<td>(Height+1) &gt;&gt; LOD</td>
<td>Surface Array Depth+1 : 0</td>
<td>MIPCount</td>
<td></td>
</tr>
<tr>
<td>SURFTYPE_3D</td>
<td>(Width+1) &gt;&gt; LOD</td>
<td>(Height+1) &gt;&gt; LOD</td>
<td>(Depth+1) &gt;&gt; LOD</td>
<td>MIPCount</td>
<td></td>
</tr>
<tr>
<td>SURFTYPE_CUBE</td>
<td>(Width+1) &gt;&gt; LOD</td>
<td>(Height+1) &gt;&gt; LOD</td>
<td>Surface Array Depth+1 : 0</td>
<td>MIPCount</td>
<td></td>
</tr>
<tr>
<td>SURFTYPE_BUFFFER</td>
<td></td>
<td>Buffer size (from combined Depth/Height/Width).</td>
<td>Undefined</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>SURFTYPE_STRBUFF</td>
<td></td>
<td>If buffer size is exactly 2^32, zero is returned in this field.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SURFTYPE_NULL 0 0 0 0

The surface is sampled using a default sampler state, indicated below. The lod parameter contains the LOD of the mip map to be sampled. If the message doesn’t include an lod parameter, the message samples from LOD 0. The parameter si contains the sample index, which is clamped to the number of samples on the surface. The v and r channel may be ignored depending on the indicated surface type. All incoming values are unsigned 32-bit integers in this mode. The u, v, and r parameters contain integer texel addresses on the LOD indicated in the parameter. The Sampler State Pointer and Sampler Index are ignored.

For these message types, the sampler state is defaulted as follows:
- min, mag, and mip filter modes are "nearest".
- All address control modes are "zero", a special mode in which any texel off the map or outside the MIP range of the surface has a value of zero in all channels, except for surface formats without an alpha channel, which return a value of one in the alpha channel.

The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFFER for the ld message.

The Surface Type of the associated surface must be SURFTYPE_2D for the ld_mcs, ld2dms, and ld2dss messages.

The Surface Format of the associated surface cannot be MONO8.

The ld_mcs message uses the MCS Base Address and MCS 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 4, or R32_UINT if Number of Multisamples is 8. This message cannot be
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Project</th>
<th>Description or Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld2dms</td>
<td></td>
<td>used on a non-multisampled surface. Otherwise, ld_mcs behaves like the ld message. If ld_mcs is issued on a surface with MCS disabled, this message returns zeros in all channels.</td>
</tr>
<tr>
<td>ld2dms_w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld_mcs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2dss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld_lz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Id            |         | The mcs parameter in the ld2dms message defines the multisample control data and is used only to sample from a multisampled surface. |
| Id2dms        |         |                           |
| Id2dms_w      |         |                           |
| ld_mcs        |         |                           |
| ld2dss        |         |                           |
| ld_lz         |         |                           |

**Note:** For the out of bound case, the following surface formats return zero in the alpha channel: R32G32B32X32_FLOAT, X32_TYPELESS_G8X24_UINT, R16G16B16X16_UNORM, R16G16B16U16X16_FLOAT, X24_TYPELESS_G8_UINT, L24X8_UNORM, L32_FLOAT, B8G8R8X8_UNORM, B8G8R8X8_UNORM_SRGB, R8G8B8X8_UNORM, R8G8B8X8_UNORM_SRGB, B10G10R10X2_UNORM, B5G6R5_UNORM, B5G6R5_UNORM_SRGB, L16_UNORM, R5G5_SNORM_B6_UNORM, L8_UNORM, L8_UNORM_SRGB, and R1_UNORM, BC4_UNORM (DXT4/5). |

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

| LOD           |         | The surface indicated in the surface state is not sampled. Instead, LOD is computed as if the surface will be sampled, using the indicated sampler state, and the clamped and unclamped LOD values are returned in the red and green channels, respectively, in FLOAT32 format. The blue and alpha channels are undefined, and can be masked to avoid returning them. LOD is computed using gradients between adjacent pixels. Three parameters are always specified, with extra parameters not needed for the surface being ignored. |
|              |         | 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. |
|              |         | LOD is not supported in SIMD4x2 mode. |
|              |         | Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. |

<p>| LOD           |         | Before HSW:C0, the Surface Format of the associated surface cannot be any UINT or SINT format. |</p>
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Project</th>
<th>Description or Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>gather4</td>
<td></td>
<td>The surface is sampled using bilinear filtering, regardless of the filtering mode specified in the sampler state. For SURFTYPE_2D, LOD is forced to zero before sampling. The samples are not filtered, but instead the four samples are returned directly in the sample's corresponding four channels as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upper left sample = alpha channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upper right sample = blue channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lower left sample = red channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lower right sample = green channel</td>
</tr>
<tr>
<td>gather4_po</td>
<td></td>
<td>Two or three parameters may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters default to 0.</td>
</tr>
<tr>
<td>(load4)</td>
<td></td>
<td>The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. If the message type is gather4_po, only SURFTYPE_2D is allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Surface Format of the associated surface cannot be MONO8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mip Mode Filter must be set to MIPFILTER_NONE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use of gather4 or gather4_po with DX9 Border Color Mode and either of the following is underfined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.</td>
</tr>
</tbody>
</table>

| gather4      |         | The gather4_po message has 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. |
|              |         | The channel selected is determined by the Gather4 Source Channel Select field in the message header. |
|gather4_po    |         | The Surface Format of the associated surface cannot be any UINT or SINT format. |
| (load4)      |         | |
| gather4      |         | Note: If Surface Format is a UINT or SINT format without alpha channel, and Gather4 Source Channel Select is alpha channel, the returned value, which should be 1, is incorrect. |
|gather4_po    |         | Note: Selecting green on R32G32_float has some erratic behavior: |
| (load4)      |         | • gather4 only on this resource returns an erroneous value if alpha is selected. |
|              |         | • gather4 + other sample operations on this resource produce erratic output. |
### Message Type | Project | Description or Restriction
--- | --- | ---
gather4 | HSW | **Note:** offu/offv are calculated in normalized space and hence subject to small truncation error.

| gather4_po | | |
| load4 | |

gather4_c | |

gather4_po_c | |

- The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. If the message type is gather4_po_c, only SURFTYPE_2D is allowed.
- The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, or R16_UNORM.
- The channel selected is determined by the Gather4 Source Channel Select field in the message header.
- Mip Mode Filter must be set to MIPFILTER_NONE.
- Use of gather4_c or gather4_po_c with DX9 Border Color Mode and either of the following is undefined:
  - Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER.
  - Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.
- Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.

| gather4_c | |
| gather4_po_c | |

- The surface is sampled using bilinear filtering, regardless of the filtering mode specified in the sampler state. For SURFTYPE_2D, LOD is forced to zero before sampling. The samples are not filtered, but instead the four samples are returned, after being compared with the ref parameter as in the sample_c message. Each texel is replaced with either white or black depending on the result of the comparison. The four samples are returned in the sample's corresponding four channels in the same mapping as the gather4 message. The offu and offv parameters in the gather4_po_c message cause offset override behavior as described in the gather4 message.

| gather4_c | |
| gather4_po_c | **Note:** offu/offv are calculated in normalized space and hence subject to small truncation error.

| sample_unorm | |

- The surface is sampled using the indicated sampler state. 32 contiguous pixels in a 8-wide by 4-high arrangement are sampled. The U and V addresses for the upper left pixel are delivered in this message along with a Delta U and Delta V parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:
  - U(x,y) = U(0,0) + DeltaU * x
  - V(x,y) = V(0,0) + DeltaV * y
- The Surface Type of the associated surface must be SURFTYPE_2D.
- The Surface Format of the associated surface must be UNORM with <= 8 bits per channel.
- The MIP Count, Depth, Surface Min LOD, Resource Min LOD, and Min Array Element of the associated surface must be 0.
- The Min and Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR.
- The Mip Mode Filter must be MIPFILTER_NONE.
The TCX and TCY Address Control Mode cannot be any of:

\[
\begin{align*}
\text{TEXCOORDMODE} & \text{\_CLAMP\_BORDER} \\
\text{TEXCOORDMODE} & \text{\_HALF\_BORDER} \\
\text{TEXCOORDMODE} & \text{\_MIRROR} \\
\text{TEXCOORDMODE} & \text{\_MIRROR\_ONCE} \\
\text{TEXCOORDMODE} & \text{\_WRAP} \\
\end{align*}
\]

DeltaU * Width of the associated surface must be less than or equal to 3.0.
DeltaV * Height of the associated surface must be less than or equal to 3.0.
Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.

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.

The surface is sampled using an optional 8x8 filter, using state defined in SAMPLER_STATE and SAMPLER_8x8_STATE.

The input consists of 64 contiguous pixels in a 16-wide by 4-high arrangement. The address control mode behaves as clamp mode. The U and V addresses for the upper left pixel are delivered in this message along with a Delta U and Delta V parameter. Given a pixel at \((x,y)\) relative to the upper left pixel (where \((0,0)\) is the upper left pixel), the \(U\) and \(V\) for that pixel are computed as follows:

\[
\begin{align*}
U(x,y) &= U(0,0) + \text{DeltaU} \times x + \text{U\text{\_2\text{nd}\_Derivative}} \times x \times (x - 1)/2 \\
V(x,y) &= V(0,0) + \text{DeltaV} \times y + \text{V\text{\_2\text{nd}\_Derivative}} \times y \times (y - 1)/2 \\
\end{align*}
\]

The Surface Type of the associated surface must be SURFTYPE_2D.
The Surface Format of the associated surface must be UNORM with \(\leq 10\) bits per channel.

\(\text{DeltaV} \times \text{Height}\) of the associated surface must be less than 16.0.

\(\text{Map Width}\) must be \(\geq 4\).

**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.

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>u</th>
<th>v</th>
<th>r</th>
<th>ai</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURFTYPE1D</td>
<td>normalized 'x' coordinate</td>
<td>unnormalized array index</td>
<td>ignored</td>
<td>ignored</td>
</tr>
<tr>
<td>SURFTYPE_2D</td>
<td>normalized 'x' coordinate</td>
<td>normalized 'y' coordinate</td>
<td>unnormalized array index</td>
<td>ignored</td>
</tr>
<tr>
<td>SURFTYPE_3D</td>
<td>normalized 'x' coordinate</td>
<td>normalized 'y' coordinate</td>
<td>normalized 'z' coordinate</td>
<td>ignored</td>
</tr>
<tr>
<td>SURFTYPE_CUBE</td>
<td>normalized 'x' coordinate</td>
<td>normalized 'y' coordinate</td>
<td>normalized 'z' coordinate</td>
<td>unnormalized array index</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>u</th>
<th>v</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURFTYPE1D</td>
<td>unnormalized 'x' coordinate</td>
<td>unnormalized array index</td>
<td>ignored</td>
</tr>
<tr>
<td>SURFTYPE_2D</td>
<td>unnormalized 'x' coordinate</td>
<td>unnormalized 'y' coordinate</td>
<td>unnormalized array index</td>
</tr>
<tr>
<td>SURFTYPE_3D</td>
<td>unnormalized 'x' coordinate</td>
<td>unnormalized 'y' coordinate</td>
<td>unnormalized 'z' coordinate</td>
</tr>
<tr>
<td>SURFTYPE_BUFFER</td>
<td>unnormalized 'x' coordinate</td>
<td>ignored</td>
<td>ignored</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 3 (lower right) Red:</strong> 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.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td><em>Supspan 0, Pixel 0 (upper left) Red</em></td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td><strong>Subspan 3, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td><strong>Subspan 3, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td><strong>Subspan 3, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td><strong>Supspan 3, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td><strong>Subspan 2, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td><strong>Subspan 2, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td><strong>Subspan 2, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td><strong>Supspan 2, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W2</td>
<td></td>
<td><strong>Subspans 1 and 0 of Green:</strong> See W0 definition for pixel locations</td>
</tr>
</tbody>
</table>
### Description

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td></td>
<td>Subspans 3 and 2 of Green: See W1 definition for pixel locations</td>
</tr>
<tr>
<td>W4</td>
<td></td>
<td>Subspans 1 and 0 of Blue: See W0 definition for pixel locations</td>
</tr>
<tr>
<td>W5</td>
<td></td>
<td>Subspans 3 and 2 of Blue: See W1 definition for pixel locations</td>
</tr>
<tr>
<td>W6</td>
<td></td>
<td>Subspans 1 and 0 of Alpha: See W0 definition for pixel locations</td>
</tr>
<tr>
<td>W7</td>
<td></td>
<td>Subspans 3 and 2 of Alpha: See W1 definition for pixel locations</td>
</tr>
<tr>
<td>W8.7:1</td>
<td></td>
<td>Reserved (not written): W8 is only delivered when <strong>Pixel Fault Mask Enable</strong> is enabled.</td>
</tr>
<tr>
<td>W8.0</td>
<td>31:16</td>
<td>Reserved: always written as 0xffff</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Pixel Null Mask</strong>: 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.</td>
</tr>
</tbody>
</table>

**Return Format = 16-bit**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:16</td>
<td><strong>Subspan 3, Pixel 3 (lower right) Red</strong>: 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 <strong>Surface Format</strong> programmed for the surface being sampled.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 3, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.6</td>
<td>31:16</td>
<td><strong>Subspan 3, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 3, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W0.5</td>
<td>31:16</td>
<td><strong>Subspan 2, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 2, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.4</td>
<td>31:16</td>
<td><strong>Subspan 2, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 2, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W0.3</td>
<td>31:16</td>
<td><strong>Subspan 1, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 1, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:16</td>
<td><strong>Subspan 1, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 1, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:16</td>
<td><strong>Subspan 0, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 0, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:16</td>
<td><strong>Subspan 0, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Subspan 0, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td><strong>Green</strong>: See W0 definition for pixel locations</td>
</tr>
</tbody>
</table>
### SIMD8/SIMD8D

**Return Format = 32-bit**

This writeback message consists of four registers, or five in the case of sample+killpix. As opposed to the SIMD16 writeback message, channels that are masked in the write channel mask are not skipped, all four channels are always returned. The masked channels, however, are not overwritten in the destination register.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 3 (lower right) Red</strong>: 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.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td><strong>Subspan 1, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 3 (lower right) Red</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 2 (lower left) Red</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 1 (upper right) Red</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td><strong>Subspan 0, Pixel 0 (upper left) Red</strong></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td><strong>Subspans 1 and 0 of Green</strong>: See W0 definition for pixel locations</td>
</tr>
<tr>
<td>W2</td>
<td></td>
<td><strong>Subspans 1 and 0 of Blue</strong>: See W0 definition for pixel locations</td>
</tr>
<tr>
<td>W3</td>
<td></td>
<td><strong>Subspans 1 and 0 of Alpha</strong>: See W0 definition for pixel locations</td>
</tr>
<tr>
<td>W4.7:1</td>
<td></td>
<td>Reserved (not written): This W4 is only delivered for the sample+killpix message type</td>
</tr>
<tr>
<td>W4.0</td>
<td>31:16</td>
<td><strong>Dispatch Pixel Mask</strong>: This field is always 0xffffffff to allow dword-based ANDing with the R0 header in the pixel shader thread.</td>
</tr>
</tbody>
</table>
DWord | Bits | Description
---|---|---
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.
| 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.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Sample 1 Alpha</strong>: 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.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td><strong>Sample 1 Blue</strong></td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td><strong>Sample 1 Green</strong></td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td><strong>Sample 1 Red</strong></td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td><strong>Sample 0 Alpha</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td><strong>Sample 0 Blue</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td><strong>Sample 0 Green</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td><strong>Sample 0 Red</strong></td>
</tr>
<tr>
<td>W1.7:1</td>
<td></td>
<td>Reserved (not written): W4 is only delivered when Pixel Fault Mask Enable is enabled.</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:2</td>
<td>Reserved: always written as 0x3fffffff</td>
</tr>
<tr>
<td></td>
<td>1:0</td>
<td><strong>Pixel Null Mask</strong>: 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.</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVB+ adds the Data Port Data Cache and the Data Cache.</td>
<td></td>
</tr>
</tbody>
</table>

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.
**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

<table>
<thead>
<tr>
<th>Project</th>
<th>Buffer Size Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Buffer Size = checked only against General State Access Upper Bound</td>
</tr>
<tr>
<td></td>
<td>When General State Access Upper Bound is zero, no bounds checking is performed.</td>
</tr>
</tbody>
</table>
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: 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:

```assembly
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.
```
Read/Write 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.

<table>
<thead>
<tr>
<th>Message</th>
<th>Applications</th>
</tr>
</thead>
</table>
| OWord Block Read/Write| • 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 | • 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 | • 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 | • 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.

<table>
<thead>
<tr>
<th>Media Boundary Pixel Mode</th>
<th>Vertical Line Stride</th>
<th>Vertical Line Stride Offset</th>
<th>Usage Model</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>normal frame</td>
<td>0000012333333</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>normal field even</td>
<td>0000022222222</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>normal field odd</td>
<td>1111133333333</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>X</td>
<td>frame / progressive</td>
<td>0000012333333</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>field even / progressive</td>
<td>0000023333333</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>field odd / progressive</td>
<td>0000133333333</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>X</td>
<td>frame / interlaced</td>
<td>0101012323232</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>field even / interlaced</td>
<td>0000022222222</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>field odd / interlaced</td>
<td>1111133333333</td>
</tr>
</tbody>
</table>
State

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

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.

Note: Data port messages may not have the End of Thread bit set in the message descriptor other than the following exeptions:

- 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

Most of the messages have an existing definition that is not expected to change. There are several new messages that are documented here.

Data Cache Data Port Message Summary

<table>
<thead>
<tr>
<th>Project</th>
<th>Message Type</th>
<th>Header Required</th>
<th>Shared Local Memory Support</th>
<th>Stateless Support</th>
<th>Address Modes</th>
<th>Vector Width</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OWord Block Read</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>global</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OWord Block Write</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>global</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Unaligned OWord Block Read</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>global</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OWord Dual Block Read</td>
<td>no for stated</td>
<td>yes for stateless</td>
<td>no</td>
<td>global + offset</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>OWord Dual Block Write</td>
<td>no for stated</td>
<td>yes for stateless</td>
<td>no</td>
<td>global + offset</td>
<td>2</td>
</tr>
<tr>
<td>Project</td>
<td>Message Type</td>
<td>Header Required</td>
<td>Shared Local Memory Support</td>
<td>Stateless Support</td>
<td>Address Modes</td>
<td>Vector Width</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------</td>
<td>----------------</td>
<td>-----------------------------</td>
<td>------------------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no for stated</td>
<td>no</td>
<td>yes</td>
<td>global +</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>DWord Scattered Read</td>
<td>yes for stateless</td>
<td></td>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>no for stated</td>
<td>no</td>
<td>yes</td>
<td>global +</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>DWord Scattered Write</td>
<td>yes for stateless</td>
<td></td>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>no for stated</td>
<td>no</td>
<td>yes</td>
<td>global +</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>Byte Scattered Read</td>
<td>yes for stateless</td>
<td></td>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>no for stated</td>
<td>no</td>
<td>yes</td>
<td>global +</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>Byte Scattered Write</td>
<td>yes for stateless</td>
<td></td>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>no for stated</td>
<td>no</td>
<td>yes</td>
<td>global +</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>Untyped Surface Read</td>
<td>yes for stateless</td>
<td></td>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Untyped Surface Write</td>
<td>no for stated</td>
<td>yes (1D only)</td>
<td>yes (1D only)</td>
<td>1D or 2D</td>
<td>2, 8, 16</td>
</tr>
<tr>
<td></td>
<td>Untyped Atomic Operation</td>
<td>no for stated</td>
<td>yes (1D only)</td>
<td>yes (1D only)</td>
<td>1D or 2D</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>Untyped Atomic Operation SIMD4x2</td>
<td>no for stated</td>
<td>yes (1D only)</td>
<td>yes (1D only)</td>
<td>1D or 2D</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Atomic Counter Operation</td>
<td>no(^2) Required for inc, dec, predec</td>
<td>no</td>
<td>no</td>
<td>implied</td>
<td>8, 16</td>
</tr>
<tr>
<td></td>
<td>Atomic Counter Operation SIMD4x2</td>
<td>no(^2) Required for inc, dec, predec</td>
<td>no</td>
<td>no</td>
<td>implied</td>
<td>2</td>
</tr>
<tr>
<td>Project</td>
<td>Message Type</td>
<td>Header Required</td>
<td>Shared Local Memory Support</td>
<td>Stateless Support</td>
<td>Address Modes</td>
<td>Vector Width</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Scratch Block Read</td>
<td>yes</td>
<td>no</td>
<td>yes (only)</td>
<td>Imm_Buf + offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scratch Block Write</td>
<td>yes</td>
<td>no</td>
<td>yes (only)</td>
<td>Imm_Buf + offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Fence</td>
<td>yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Typed Surface Read</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1D, 2D, 3D, 4D</td>
<td>2, 8</td>
<td></td>
</tr>
<tr>
<td>Typed Surface Write</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1D, 2D, 3D, 4D</td>
<td>2, 8</td>
<td></td>
</tr>
<tr>
<td>Typed Atomic Operation</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1D, 2D, 3D, 4D</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Typed Atomic Operation</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1D, 2D, 3D, 4D</td>
<td>2, 8</td>
<td></td>
</tr>
<tr>
<td>Media Block Read</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>2D</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Media Block Write</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>2D</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**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.
Note 2 in the table above: **Note:** For Atomic Counter OPS other than INC, DEC, and PREDEC, the header is forbidden and not optional as indicated in the table.

<table>
<thead>
<tr>
<th>Project:</th>
<th></th>
<th>HSW</th>
</tr>
</thead>
</table>

### Render Cache Data Port Message Summary

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Header Required</th>
<th>Address Modes</th>
<th>Vector Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media Block Read (legacy)</td>
<td>yes</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>Media Block Write (non-IECP)</td>
<td>yes</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>Render Target Write</td>
<td>no</td>
<td>2D + RTAI</td>
<td>8, 16</td>
</tr>
<tr>
<td>Memory Fence</td>
<td>yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Header Present. If set, indicates that the message includes the header. Refer to Render Target Write message section for more details on this field. Programming Notes: The header must be present unless the message type is Render Target Write Format = Enable</td>
</tr>
<tr>
<td>18</td>
<td>Ignored</td>
</tr>
<tr>
<td>17:16</td>
<td>Ignored</td>
</tr>
<tr>
<td>15:13</td>
<td>Message Type</td>
</tr>
<tr>
<td>15:13</td>
<td>Message Type</td>
</tr>
<tr>
<td>16:13</td>
<td>Message Type</td>
</tr>
</tbody>
</table>

- **Data Port Sampler Cache**
  - Bit 19: Header Present
  - Bit 18: Ignored
  - Bit 17:16: Ignored
  - Bit 15:13: Message Type

- **Data Port Constant Cache**
  - Bit 19: Header Present
  - Bit 18: Ignored
  - Bit 17:16: Ignored

- **Data Port Render Cache**
  - Bit 19: Header Present
  - Bit 18: Ignored
  - Bit 17: Send Write Commit Message. Indicates that a write commit message will be sent back to the thread when the write has been committed. See section **Write Commit** for more details. This field is ignored on read message types. Format = Enable
  - Bit 15:13: Message Type

- **Message Type**
  - 000: OWord Block Read
  - 010: OWord Dual Block Read
  - 100: Media Block Read
  - 0000: OWord Block Read
  - 0001: Render Target UNORM Read
  - 0010: OWord Dual Block Read
  - 0100: Media Block Read
  - 0101: Unaligned OWord Block Read
101: Unaligned OWord Block Read  
110: DWord Scattered Read  
All other encodings are reserved.

0110: DWord Scattered Read  
0111: DWord Atomic write message  
1000: OWord Block Write  
1001: OWord Dual Block Write  
1010: Media Block Write  
1011: DWord Scattered Write  
1100: Render Target Write  
1101: Streamed Vertex Buffer Write  
1110: Render Target UNORM Write  
All other encodings are reserved.

12:8 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. A binding table index of 255 indicates that a stateless model is to be used. The stateless model is allowed only with the render cache data port. Refer to section 2.2.2 for details on the stateless model.

Format = U8  
Range = [0,255]

Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 19  | Header Present. If set, indicates that the message includes the header. Programming Notes:  
For the Render cache Data Port, the header must be present for the following message types: Memory Fence Media block read Media block write  
For 3d RT reads and writes, header is optionally present.  
For the Sampler Cache Data Port, the header must be present for the following message types: Unaligned OWord Block Read Media block read |  | |
### Sampler Cache Data Port

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Ignored</td>
<td>18</td>
<td>Ignored</td>
</tr>
<tr>
<td>17:14</td>
<td>Message Type</td>
<td>17:14</td>
<td>Message Type</td>
</tr>
<tr>
<td></td>
<td>0000: Read Surface Info</td>
<td></td>
<td>0100: Media Block Read (legacy)</td>
</tr>
<tr>
<td></td>
<td>0001: Unaligned OWord Block Read</td>
<td></td>
<td>0111: Memory Fence</td>
</tr>
<tr>
<td></td>
<td>0100: Media Block Read</td>
<td></td>
<td>1010: Media Block Write (non-IÉCP)</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
<td></td>
<td>1100: Render Target Write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

#### Message Specific Control
Refer to the specific message section for the definition of these bits.

### Constant Cache Data Port

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Header Present. If set, indicates that the message includes the header.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Programming Notes:

- For the Data Cache Data Port*, the header must be present for the following message types:
  - OWord Block Read/Write
  - Unaligned OWord Block Read
  - Memory Fence
  - Scratch read/write
  - Typed read/write/atomics
  - Media block read/write

- For the Constant Cache Data Port, the header must be present for the following message types:
  - OWord Block Read/Write
  - Unaligned OWord Block Read

### Data Cache Data Port0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Ignored</td>
<td>18</td>
<td>Category 1: Scratch Block Read/Write messages</td>
</tr>
</tbody>
</table>

### Data Cache Data Port1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Ignored</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>Legacy DAP-DC messages</td>
</tr>
<tr>
<td>17:14</td>
<td>Message Type</td>
</tr>
<tr>
<td>0</td>
<td>Category=0 (legacy dataport)</td>
</tr>
<tr>
<td></td>
<td>Message Type</td>
</tr>
<tr>
<td>0</td>
<td>OWord Block Read</td>
</tr>
<tr>
<td>0</td>
<td>Unaligned OWord Block Read</td>
</tr>
<tr>
<td>0</td>
<td>OWord Dual Block Read</td>
</tr>
<tr>
<td>0</td>
<td>DWord Scattered Read</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td>17:14</td>
<td>Category=1 (scratch)</td>
</tr>
<tr>
<td>17:14</td>
<td>Type;</td>
</tr>
<tr>
<td>16</td>
<td>Type;</td>
</tr>
<tr>
<td>0</td>
<td>OWord, 1= Dword</td>
</tr>
<tr>
<td>15</td>
<td>Invalidate after read;</td>
</tr>
<tr>
<td>14</td>
<td>&lt;Reserved, mbz&gt;</td>
</tr>
<tr>
<td>13:12</td>
<td>Block Size</td>
</tr>
<tr>
<td>11</td>
<td>4 registers</td>
</tr>
<tr>
<td>10</td>
<td>&lt;reserved&gt;</td>
</tr>
<tr>
<td>01</td>
<td>2 registers</td>
</tr>
<tr>
<td>00</td>
<td>1 register</td>
</tr>
<tr>
<td>11:0</td>
<td>Addr offset (Hword based)</td>
</tr>
</tbody>
</table>

**Message Specific Control.** Refer to the specific message section for the definition of these bits.

**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 "Shared Local Memory" 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 section "Stateless Model" section for details on the stateless model.

253: An alias for Stateless
252: An alias for Stateless
251: An alias for Stateless
250: An alias for Stateless

| Format = U8 |
| Range = [0,255] |

[DevHSW+] 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.

<table>
<thead>
<tr>
<th>Binding table index</th>
<th>Coherency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>Locally Coherent</td>
</tr>
<tr>
<td>253</td>
<td>Non-Coherent</td>
</tr>
<tr>
<td>252</td>
<td>Globally Coherent</td>
</tr>
<tr>
<td>251</td>
<td>LLC Coherent</td>
</tr>
<tr>
<td>250</td>
<td>L3UC</td>
</tr>
</tbody>
</table>

**Programming Restriction:** When using BTI = 253, SW must ensure that 2 threads do not both access the same cache line (64B).

**Programming Note:** If the stateless access falls between the LLC Coherent Base Address and the LLC Coherent Upper bound and the BTI is not equal to 250, then the access will be forced to take on the LLC coherent attribute and behave accordingly.

**Notes:** Binding table indexes 250-253 are not implemented. SW should treat these as reserved and have the binding table for these entries point to a surface state of type SURFTYPE_NULL.
# Message Header

This header applies to the following data port messages:

<table>
<thead>
<tr>
<th>Project</th>
<th>Data Port Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OWord Block Read/Write</td>
</tr>
<tr>
<td></td>
<td>Unaligned OWord Block Read</td>
</tr>
<tr>
<td></td>
<td>OWord Dual Block Read/Write</td>
</tr>
<tr>
<td></td>
<td>DWord Scattered Read/Write</td>
</tr>
<tr>
<td></td>
<td>Byte Scattered Read/Write</td>
</tr>
<tr>
<td></td>
<td>Scratch Block Read/Write</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.5</td>
<td>31:10</td>
<td><strong>Immediate Buffer Base Address.</strong> 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]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9:8</td>
<td>Ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Dispatch ID.</strong> 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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:4</td>
<td>Ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td><strong>Per Thread Scratch Space</strong> 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</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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).

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:0</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The only surface type allowed is SURFTYPE_BUFFER.</td>
</tr>
<tr>
<td></td>
<td>The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.</td>
</tr>
<tr>
<td></td>
<td>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 a DWord is out-of-bounds then the whole DWord is considered out-of-bounds.</td>
</tr>
<tr>
<td></td>
<td>The surface cannot be tiled.</td>
</tr>
<tr>
<td></td>
<td>The surface base address must be OWord-aligned.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Render Cache Read Write Mode</strong> field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Stateless Render Cache Read-Write Mode</strong> 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.</td>
</tr>
</tbody>
</table>

### Applications:

- Constant buffer reads of a single constant or multiple contiguous constants.
- Scratch space reads/writes where the index for each pixel/vertex is the same.
- Block constant reads, scratch memory reads/writes for media.

### 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

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
### Project Bits Description

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Invalidate After Read Enable. This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>10:8</td>
<td>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. Programming Note: The 6 OWord block size is valid only with Data Port Constant Cache.</td>
<td></td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7:4</td>
<td>127.0</td>
<td>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.</td>
</tr>
<tr>
<td>M1.3:0</td>
<td>127.0</td>
<td>OWord[Offset]</td>
</tr>
<tr>
<td>M2.7:4</td>
<td>127.0</td>
<td>OWord[Offset+3]</td>
</tr>
<tr>
<td>M2.3:0</td>
<td>127.0</td>
<td>OWord[Offset+2]</td>
</tr>
<tr>
<td>M3.7:4</td>
<td>127.0</td>
<td>OWord[Offset+5]</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3.3:0</td>
<td>127:0</td>
<td>OWord[Offset+4]</td>
</tr>
<tr>
<td>M4.7:4</td>
<td>127:0</td>
<td>OWord[Offset+7]</td>
</tr>
<tr>
<td>M4.3:0</td>
<td>127:0</td>
<td>OWord[Offset+6]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>The only surface type allowed is SURFTYPE_BUFFER.</td>
<td></td>
</tr>
<tr>
<td>The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.</td>
<td></td>
</tr>
<tr>
<td>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 a DWord is out-of-bounds then the whole DWord is considered out-of-bounds.</td>
<td></td>
</tr>
<tr>
<td>The surface cannot be tiled.</td>
<td></td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>12:11</td>
<td></td>
<td>Ignored</td>
</tr>
</tbody>
</table>
| 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.

<table>
<thead>
<tr>
<th>DWORD</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:4</td>
<td>127:0</td>
<td>*<em>OWord1 = <em>(OWord0 + 1).</em></em> If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord0 will appear in this location</td>
</tr>
<tr>
<td>W0.3:0</td>
<td>127:0</td>
<td>OWord0 = Buffer[Global Offset]</td>
</tr>
<tr>
<td>W1.7:4</td>
<td>127:0</td>
<td>OWord3 = *(OWord2 + 1)</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>W1.3:0 127:0</td>
<td></td>
<td>OWord2 = *(&amp;OWord1 + 1)</td>
</tr>
<tr>
<td>W2.7:4 127:0</td>
<td></td>
<td>OWord5 = *(&amp;OWord4 + 1)</td>
</tr>
<tr>
<td>W2.3:0 127:0</td>
<td></td>
<td>OWord4 = *(&amp;OWord3 + 1)</td>
</tr>
<tr>
<td>W3.7:4 127:0</td>
<td></td>
<td>OWord7 = *(&amp;OWord6 + 1)</td>
</tr>
<tr>
<td>W3.3:0 127:0</td>
<td></td>
<td>OWord6 = *(&amp;OWord5 + 1)</td>
</tr>
</tbody>
</table>

**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.

**Project:**

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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The only surface type allowed is SURFTYPE_BUFFER.</td>
</tr>
<tr>
<td></td>
<td>The surface format is ignored; data is returned from the constant buffer to</td>
</tr>
<tr>
<td></td>
<td>the GRF without format conversion.</td>
</tr>
<tr>
<td></td>
<td>The surface is treated as a 1-dimensional surface. The element size (pitch)</td>
</tr>
<tr>
<td></td>
<td>times the number of elements is used to determine the size of the buffer</td>
</tr>
<tr>
<td></td>
<td>for out-of-bounds checking if using the surface state model. Out of bounds</td>
</tr>
<tr>
<td></td>
<td>checking is done at DWord granularity; if any part of a DWord is</td>
</tr>
<tr>
<td></td>
<td>out-of-bounds then the whole DWord is considered out-of-bounds.</td>
</tr>
<tr>
<td></td>
<td>The surface cannot be tiled.</td>
</tr>
<tr>
<td></td>
<td>The surface base address must be OWord-aligned.</td>
</tr>
<tr>
<td></td>
<td>The Render Cache Read Write Mode field in SURFACE_STATE must be set to</td>
</tr>
<tr>
<td></td>
<td>read/write mode when using this message with the render cache in the</td>
</tr>
<tr>
<td></td>
<td>surface state model.</td>
</tr>
<tr>
<td></td>
<td>The Stateless Render Cache Read-Write Mode field in the SVG_WORK_CTL</td>
</tr>
<tr>
<td></td>
<td>register must be set to read/write mode when using this message with the</td>
</tr>
<tr>
<td></td>
<td>render cache in the stateless model.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
<td><strong>Invalidate After Read Enable.</strong> This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable</td>
</tr>
<tr>
<td></td>
<td>12:10</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>9:8</td>
<td><strong>Block Size.</strong> Specifies the number of OWords in each block to be read or written: 00: 1 OWord 10: 4 OWords All other encodings are reserved.</td>
</tr>
</tbody>
</table>

**Message Payload**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td><strong>Block Offset 1.</strong> Specifies the OWord offset of OWord Block 1 into the surface. Format = U32 Range = [0,0FFFFFFFh]</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td><strong>Block Offset 0</strong></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.7:4</td>
<td>127:0</td>
<td>OWord[Offset1]</td>
</tr>
<tr>
<td>M2.3:0</td>
<td>127:0</td>
<td>OWord[Offset0]</td>
</tr>
<tr>
<td>M3.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+1]</td>
</tr>
<tr>
<td>M3.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+1]</td>
</tr>
<tr>
<td>M4.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+2]</td>
</tr>
<tr>
<td>M4.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+2]</td>
</tr>
<tr>
<td>M4.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+3]</td>
</tr>
<tr>
<td>M4.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+3]</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:4</td>
<td>127:0</td>
<td>OWord[Offset1]</td>
</tr>
<tr>
<td>W0.3:0</td>
<td>127:0</td>
<td>OWord[Offset0]</td>
</tr>
<tr>
<td>W1.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+1]</td>
</tr>
<tr>
<td>W1.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+1]</td>
</tr>
<tr>
<td>W2.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+2]</td>
</tr>
<tr>
<td>W2.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+2]</td>
</tr>
<tr>
<td>W3.7:4</td>
<td>127:0</td>
<td>OWord[Offset1+3]</td>
</tr>
<tr>
<td>W3.3:0</td>
<td>127:0</td>
<td>OWord[Offset0+3]</td>
</tr>
</tbody>
</table>

Media Block Read/Write

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The target cache cannot be the data cache.</td>
</tr>
<tr>
<td></td>
<td>The surface base address must be 32-byte aligned.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td>The write form of this message has the additional restriction that both X Offset and Block Width must be DWord-aligned.</td>
</tr>
<tr>
<td></td>
<td>Pitch must be a multiple of 64 bytes when the surface is linear.</td>
</tr>
</tbody>
</table>

**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 B0B0B0B1, 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 \textbf{Y1U0Y0V0}, and to replicate the right boundary, we get \textbf{Y0U0Y1V0}.

For a boundary DWord U0Y0V0Y1, to replicate the left boundary, we get \textbf{U0Y0V0Y1}, and to replicate the right boundary, we get \textbf{U0Y1V0Y1}.

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.

When Color Processing Enable is set to 1 and the IECP output surface to be written is NV12 format (R16_UNORM surface format 0x10A, should be used if the output surface is NV12 format).

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 mode
- 4 pixels wide for 444 8-bit mode
- 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).
<table>
<thead>
<tr>
<th>Project</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>13</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Vertical Line Stride Override</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies whether the <strong>Vertical Line Stride</strong> and <strong>Vertical Line Stride Offset</strong> fields in the surface state should be replaced by bits 9 and 8 below.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<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>
</thead>
<tbody>
<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>
</tbody>
</table>

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).

0: Use parameters in the surface state and ignore bits 9:8.
1: Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset.
### Project Bit Description

<table>
<thead>
<tr>
<th>Project</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
|         | 8   | **Override Vertical Line Stride Offset**  
|         |     | Specifies the offset of the initial line from the beginning of the buffer. Ignored when **Override Vertical Line Stride** is 0.  
|         |     | Format = U1 in lines of initial offset (when Vertical Line Stride == 1). |

### Message Header

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.5</td>
<td>31:8</td>
<td>Ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>FFTID.</strong> 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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| M0.3  | 31:5 | **Color Processing State Pointer.** Defines the pointer to COLOR_PROCESSING_STATE. Ignored on read messages and when **Color Processing Enable** is not set. This pointer is relative to the **General State Base Address**.  
**Programming Note:** This pointer is *not* delivered via state variables like most other pointers are delivered. It must be delivered via another software-defined mechanism such as CURBE.  
Format = GeneralStateOffset[31:5] |         |          |
|       | 4    | **Message Mode.** This field selects the mode of this message as follows:  
0: 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.  
1: PIXEL_MARGIN. The **Pixel Mask** field is set in M0.2. The **Block Height** and **Block Width** are not explicitly set but behave as if they are set to 4 rows and 32 bytes, respectively.  
**Programming Note:** Only NORMAL mode is allowed for Block width > 32 bytes.  
For the **Sampler Cache Data Port**, this field is also ignored, behaving as if always set to NORMAL. |         |          |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:2</td>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><strong>Area of Interest.</strong> This field controls whether the statistic for the luma pixels is collected at VSC for ACE histogram. This field is effective only when the state variable <code>Full_image_histogram</code> is disabled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following M0.2 definition applies only if the **Message Mode** field is set to NORMAL:

<table>
<thead>
<tr>
<th>M0.2</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:29</td>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 28:24| **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. **Programming Notes:**

*Sub-Register Offset* must be aligned to `BasePitch` (therefore will be a multiple of DWords as well).

When `Register Pitch Control` = 0, **Sub-Register Offset** must align to `BasePitch*Block Height` and the output fits in a single GRF register.

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.

**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.

This field must be zero for Render Cache Data Port.

*Format = US*

*Range = [0, 28] (Only a multiple of `BasePitch`, including 0, is valid.)*

<p>| 21:16 | <strong>Block Height.</strong> Height in rows of block being accessed. <strong>Programming Note:</strong> The Block Height is restricted to the following maximum values depending on the Block Width: | | | |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Block Width</strong>&lt;br&gt;(bytes)</td>
<td><strong>Maximum Block Height</strong>&lt;br&gt;(rows)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-4</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5-8</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9-16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17-32</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Format = U6
Range = [0,63] representing 1 to 64 rows

15:10  Ignored

9:8  **Register Pitch Control** This field controls the register pitch for a media block read message. This field is ignored (reserved) for a media block write message.

**Programming Notes:**
This field must be zero for Render Cache Data Port.

**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).

Define **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 term of **BasePitch** as the following:
Range = [0,3] representing 1 to 4 **BasePitch**

7:5  Ignored

4:0  **Block Width.** Width in bytes of the block being accessed.

**Programming Note:** Must be DWord-aligned for the write form of the message.

Range = [0,31] representing 1 to 32 bytes

The following MO.2 definition applies only if the Message Mode field is set to PIXEL_MASK:

MO.2 31:0  **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.

The bits in this mask correspond to the pixels (DWords) as follows:
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:0</td>
<td><strong>Y offset.</strong> The Y offset of the upper left corner of the block into the surface. Format = S31</td>
</tr>
<tr>
<td>2</td>
<td>31:0</td>
<td><strong>X offset.</strong> The X offset of the upper left corner of the block into the surface. Must be DWord-aligned (Bits 1:0 MBZ) for the write form of the message. The <strong>X offset</strong> 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. Format = S31</td>
</tr>
<tr>
<td>8</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>31:0</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:** If **Message Mode** is set to PIXEL_MASK, this field must be a multiple of 4.

**Programming Note:** If **Message Mode** is set to PIXEL_MASK, this field must be a multiple of 32.

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

<table>
<thead>
<tr>
<th>Block Height for given block width, pitch control, subreg offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>block width pitch control</td>
</tr>
<tr>
<td>1-4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>5-8</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>9-16</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### Block Height for given block width, pitch control, subreg offsets

<table>
<thead>
<tr>
<th>block width</th>
<th>pitch control</th>
<th>sub-register offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1-16</td>
<td>illegal illegal illegal 1-16 illegal illegal illegal</td>
</tr>
<tr>
<td>7-32</td>
<td>00</td>
<td>1-8 illegal illegal illegal illegal illegal illegal</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>illegal illegal illegal illegal illegal illegal illegal</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>illegal illegal illegal illegal illegal illegal illegal</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1-8 illegal illegal illegal illegal illegal illegal illegal</td>
</tr>
</tbody>
</table>

### Message Payload (Write)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1:n</td>
<td></td>
<td><strong>Write Data.</strong> The format of the write data depends on the <em>Block Height</em> and <em>Block Width</em>. 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 <em>Block Width</em>.</td>
</tr>
</tbody>
</table>

If *Color Processing Enable* is enabled, the write data is divided into pixels according to the **Message Format** field. The fields within each pixel are defined below. For the 4:2:2 modes, each pixel position includes channels for two pixels.

<table>
<thead>
<tr>
<th>Message Format</th>
<th>31:24</th>
<th>23:16</th>
<th>15:8</th>
<th>7:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>YUV 4:2:2, 8 bits per channel</td>
<td>Cr (V)</td>
<td>right pixel lum (Y1)</td>
<td>Cb (U)</td>
<td>left pixel lum (Y0)</td>
</tr>
<tr>
<td>YUV 4:4:4, 8 bits per channel</td>
<td>alpha (A)</td>
<td>luminance (Y)</td>
<td>Cb (U)</td>
<td>Cr (V)</td>
</tr>
<tr>
<td></td>
<td>63:48</td>
<td>47:32</td>
<td>31:16</td>
<td>15:0</td>
</tr>
<tr>
<td>YUV 4:2:2, 16 bits per channel</td>
<td>Cr (V)</td>
<td>right pixel lum (Y1)</td>
<td>Cb (U)</td>
<td>left pixel lum (Y0)</td>
</tr>
<tr>
<td>YUV 4:4:4, 16 bits per channel</td>
<td>alpha (A)</td>
<td>Cr (V)</td>
<td>luminance (Y)</td>
<td>Cb (U)</td>
</tr>
</tbody>
</table>

### Writeback Message (Read)

<table>
<thead>
<tr>
<th>Project</th>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0:n</td>
<td>31:0</td>
<td>Read Data. The format of the read data depends on the <em>Block Height</em>, <em>Block Width</em>, <em>Register Pitch Control</em>, and <em>Sub-Register Offset</em>. The data is aligned to the <em>Sub-Register Offset</em> of the first register, and the register pitch is set to one or more <em>BasePatch</em>.</td>
<td></td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The only surface type allowed is SURFTYPE_BUFFER.</td>
</tr>
<tr>
<td></td>
<td>The surface format is ignored; data is returned from the constant buffer to the GRF without format conversion.</td>
</tr>
<tr>
<td></td>
<td>The surface cannot be tiled.</td>
</tr>
<tr>
<td></td>
<td>The surface base address must be DWord-aligned.</td>
</tr>
<tr>
<td></td>
<td>Writes to overlapping addresses have undefined write ordering.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Render Cache Read Write Mode</strong> field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Stateless Render Cache Read-Write Mode</strong> 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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The message header is no longer required for the <strong>OWord DWord Scattered Read/Write</strong> messages if sent to the data cache data port. If header is not sent, the <strong>Global Offset</strong> field is assumed to be zero. The header is required, however, if the binding table index is 255 (stateless model), as the <strong>Immediate Buffer Base Address</strong> field is required.</td>
</tr>
<tr>
<td></td>
<td>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 a DWord granularity; if any part of the DWord is out-of-bounds then the whole DWord is considered out-of-bounds.</td>
</tr>
</tbody>
</table>

### Applications:

- SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message)
- SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message)
- General purpose DWord scatter/gathering, used by media

### 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.
<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>13</td>
<td><strong>Invalid After Read Enable.</strong> If enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>Ignored</td>
</tr>
<tr>
<td>11:10</td>
<td>11:10</td>
<td>Ignored</td>
</tr>
<tr>
<td>9:8</td>
<td>9:8</td>
<td><strong>Block Size.</strong> Specifies the number of DWords read or written: 10: 8 DWords 11: 16 DWords All other encodings are reserved.</td>
</tr>
</tbody>
</table>
## Message Payload

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| M1.7  | 31:0 | Offset 7. Specifies the DWord offset of DWord 7 into the surface.  
|       |      | Format = U32  
|       |      | Range = [0,3FFFFFFFh] |
| 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 **Offset** referred to below is the **Global Offset** added to the corresponding **Offset** and is in units of DWords. The **DWord** array index is also in units of DWords.

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

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

Message Descriptor

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

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

**Message Payload**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| M1.7  | 31:0 | Offset 7.  
Specifies the DWord offset of DWord 7 into the surface.  
Format = U32  
Range = [0,3FFFFFFFh] |
| 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 Offset\textsuperscript{n} referred to below is the Global Offset added to the corresponding Offset\textsuperscript{n} and is in units of DWords. The DWord array index is also in units of DWords.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>DWord[Offset7] Src0</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>DWord[Offset6] Src0</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>DWord[Offset5] Src0</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>DWord[Offset4] Src0</td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td>DWord[Offset3] Src0</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td>DWord[Offset2] Src0</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td>DWord[Offset1] Src0</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td>DWord[Offset0] Src0</td>
</tr>
<tr>
<td>M3.7</td>
<td>31:0</td>
<td>DWord[Offset7] Src1</td>
</tr>
<tr>
<td>M3.6</td>
<td>31:0</td>
<td>DWord[Offset6] Src1</td>
</tr>
<tr>
<td>M3.5</td>
<td>31:0</td>
<td>DWord[Offset5] Src1</td>
</tr>
<tr>
<td>M3.4</td>
<td>31:0</td>
<td>DWord[Offset4] Src1</td>
</tr>
<tr>
<td>M3.3</td>
<td>31:0</td>
<td>DWord[Offset3] Src1</td>
</tr>
<tr>
<td>M3.2</td>
<td>31:0</td>
<td>DWord[Offset2] Src1</td>
</tr>
<tr>
<td>M3.1</td>
<td>31:0</td>
<td>DWord[Offset1] Src1</td>
</tr>
<tr>
<td>M3.0</td>
<td>31:0</td>
<td>DWord[Offset0] Src1</td>
</tr>
</tbody>
</table>
Writeback Message

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

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>DWord[Offset7]</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>DWord[Offset6]</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>DWord[Offset5]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>DWord[Offset4]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>DWord[Offset3]</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>DWord[Offset2]</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>DWord[Offset1]</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>DWord[Offset0]</td>
</tr>
</tbody>
</table>

Byte Scattered ReadWrite

These messages are supported on IVB+ 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

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The only surface type allowed is SURFTYPE_BUFFER.</td>
</tr>
<tr>
<td></td>
<td>The surface format is ignored; data is returned from the buffer to the GRF without format conversion.</td>
</tr>
<tr>
<td></td>
<td>The surface cannot be tiled.</td>
</tr>
<tr>
<td></td>
<td>The surface base address must be DWord-aligned.</td>
</tr>
<tr>
<td></td>
<td>Writes to overlapping addresses have undefined write ordering.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The stateless model is supported. Bounds checking for a stateless message is 4GB overflow and &lt; General State upper bound.</td>
</tr>
<tr>
<td></td>
<td>For byte scattered read and write the buffer size must be a multiple of 4 bytes.</td>
</tr>
</tbody>
</table>

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**

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

**Message Payload**

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

**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 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3.7</td>
<td>31:0</td>
<td>Data[Offset7]</td>
</tr>
<tr>
<td>M3.6</td>
<td>31:0</td>
<td>Data[Offset6]</td>
</tr>
<tr>
<td>M3.5</td>
<td>31:0</td>
<td>Data[Offset5]</td>
</tr>
<tr>
<td>M3.4</td>
<td>31:0</td>
<td>Data[Offset4]</td>
</tr>
<tr>
<td>M3.3</td>
<td>31:0</td>
<td>Data[Offset3]</td>
</tr>
<tr>
<td>M3.2</td>
<td>31:0</td>
<td>Data[Offset2]</td>
</tr>
<tr>
<td>M3.1</td>
<td>31:0</td>
<td>Data[Offset1]</td>
</tr>
<tr>
<td>M3.0</td>
<td>31:0</td>
<td>Data[Offset0]</td>
</tr>
<tr>
<td>M4.7</td>
<td>31:0</td>
<td>Data[Offset15]. This message register is included only if the SIMD Mode is SIMD16.</td>
</tr>
<tr>
<td>M4.6</td>
<td>31:0</td>
<td>Data[Offset14]</td>
</tr>
<tr>
<td>M4.5</td>
<td>31:0</td>
<td>Data[Offset13]</td>
</tr>
<tr>
<td>M4.4</td>
<td>31:0</td>
<td>Data[Offset12]</td>
</tr>
<tr>
<td>M4.3</td>
<td>31:0</td>
<td>Data[Offset11]</td>
</tr>
<tr>
<td>M4.2</td>
<td>31:0</td>
<td>Data[Offset10]</td>
</tr>
<tr>
<td>M4.1</td>
<td>31:0</td>
<td>Data[Offset9]</td>
</tr>
<tr>
<td>M4.0</td>
<td>31:0</td>
<td>Data[Offset8]</td>
</tr>
</tbody>
</table>
Writeback Message (Read)

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

The Offsets 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.

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

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Supported via the data cache data port.</td>
</tr>
</tbody>
</table>

**Programming Restriction:** The Vertical stride & Vertical Offset fields of the surface state object are only supported for 2D non-array surfaces.
- **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. All SLM accesses are in this category.
- **Structured buffer (untyped).** These surfaces are of type SURFTYPE_STRBUF and have a surface format of RAW. Supported via the data cache data port.

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:

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>new_dst</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP_AND</td>
<td>old_dst &amp; src0</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_OR</td>
<td>old_dst</td>
<td>src0</td>
</tr>
<tr>
<td>AOP_XOR</td>
<td>old_dst ^ src0</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_MOV</td>
<td>src0</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_INC</td>
<td>old_dst + 1</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_DEC</td>
<td>old_dst</td>
<td>src0</td>
</tr>
<tr>
<td>AOP_ADD</td>
<td>old_dst</td>
<td>src0</td>
</tr>
<tr>
<td>AOP_SUB</td>
<td>old_dst - src0</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_REVSUB</td>
<td>src0 - old_dst</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_IMAX</td>
<td>imax(old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_IMIN</td>
<td>imin(old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_UMAX</td>
<td>umax(old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_UMIN</td>
<td>umin(old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_CMPWR</td>
<td>(src0 == old_dst) ? src1 : old_dst</td>
<td>old_dst</td>
</tr>
<tr>
<td>AOP_PREDEC</td>
<td>old_dst</td>
<td>src0</td>
</tr>
<tr>
<td>AOP_CMPWR8B</td>
<td>(src08B == old_dst8B) ? src18B : old_dst8B</td>
<td>old_dst8B</td>
</tr>
</tbody>
</table>

**Programming Note:** src08B is 8 bytes, src18B is 8 bytes, and old_dst8B is 8 bytes in length.

**Programming Note:** AOP_CMPWR8B is not supported for SLM.

**Programming Note:** AOP_CMPWR8B addresses must be QWord-aligned.

**Note:** 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.

**Restrictions:**

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Surface Format Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R16G16B16A16_UINT</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_UINT</td>
<td></td>
</tr>
<tr>
<td>R16G16_UINT</td>
<td></td>
</tr>
<tr>
<td>R32_SINT</td>
<td></td>
</tr>
<tr>
<td>R32_UINT</td>
<td></td>
</tr>
<tr>
<td>R32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>R8G8_UINT</td>
<td></td>
</tr>
<tr>
<td>R16_UINT</td>
<td></td>
</tr>
<tr>
<td>R8_UINT</td>
<td></td>
</tr>
</tbody>
</table>

## Surface Format for Typed Surface Reads

<table>
<thead>
<tr>
<th>Project</th>
<th>Surface Format Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R32G32B32A32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_SINT</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_UINT</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UNORM</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SNORM</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SINT</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UINT</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_FLOAT</td>
<td></td>
</tr>
<tr>
<td>R32G32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>R32G32_SINT</td>
<td></td>
</tr>
<tr>
<td>R32G32_UINT</td>
<td></td>
</tr>
<tr>
<td>B8G8R8A8_UNORM</td>
<td></td>
</tr>
<tr>
<td>R10G10B10A2_UNORM</td>
<td></td>
</tr>
<tr>
<td>R10G10B10A2_UINT</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_UNORM</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_SNORM</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_SINT</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_UINT</td>
<td></td>
</tr>
<tr>
<td>R16G16_UNORM</td>
<td></td>
</tr>
<tr>
<td>R16G16_SNORM</td>
<td></td>
</tr>
<tr>
<td>R16G16_SINT</td>
<td></td>
</tr>
<tr>
<td>R16G16_UINT</td>
<td></td>
</tr>
<tr>
<td>R16G16_FLOAT</td>
<td></td>
</tr>
<tr>
<td>B10G10R10A2_UNORM</td>
<td></td>
</tr>
<tr>
<td>R11G11B10_FLOAT</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Surface Format Name</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>R32_SINT</td>
<td></td>
</tr>
<tr>
<td>R32_UINT</td>
<td></td>
</tr>
<tr>
<td>R32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>B5G6R5_UNORM</td>
<td></td>
</tr>
<tr>
<td>B5G5R5A1_UNORM</td>
<td></td>
</tr>
<tr>
<td>B4G4R4A4_UNORM</td>
<td></td>
</tr>
<tr>
<td>R8G8_UNORM</td>
<td></td>
</tr>
<tr>
<td>R8G8_SNORM</td>
<td></td>
</tr>
<tr>
<td>R8G8_SINT</td>
<td></td>
</tr>
<tr>
<td>R8G8_UINT</td>
<td></td>
</tr>
<tr>
<td>R16_UNORM</td>
<td></td>
</tr>
<tr>
<td>R16_SNORM</td>
<td></td>
</tr>
<tr>
<td>R16_SINT</td>
<td></td>
</tr>
<tr>
<td>R16_UINT</td>
<td></td>
</tr>
<tr>
<td>R16_FLOAT</td>
<td></td>
</tr>
<tr>
<td>B5G5R5X1_UNORM</td>
<td></td>
</tr>
<tr>
<td>R8_UNORM</td>
<td></td>
</tr>
<tr>
<td>R8_SNORM</td>
<td></td>
</tr>
<tr>
<td>R8_SINT</td>
<td></td>
</tr>
<tr>
<td>R8_UINT</td>
<td></td>
</tr>
<tr>
<td>A8_UNORM</td>
<td></td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The stateless model support is limited to untyped messages. Furthermore, they are treated as SURFTYPE_BUFFER and <strong>Surface Format</strong> of RAW. The bounds checking for the stateless message is 4GB overflow and &lt; General State upper bound.</td>
</tr>
</tbody>
</table>

**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.

**Note:** The **Typed Surface Read** message returns 0 in all channels for out-of-bounds accesses.

**Programming Restriction:** Writes to overlapping addresses have undefined write ordering.
### SIMD Mode, Surface Category, and Message Type Combinations Supported

<table>
<thead>
<tr>
<th>SIMD Mode</th>
<th>Surface Category</th>
<th>Message Type</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD16</td>
<td>Untyped</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>SIMD16</td>
<td>Untyped</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>SIMD16</td>
<td>Untyped</td>
<td>Atomic</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Untyped</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Untyped</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Untyped</td>
<td>Atomic</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Typed</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Typed</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>SIMD8</td>
<td>Typed</td>
<td>Atomic</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Untyped</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Untyped</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Untyped</td>
<td>Atomic</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Typed</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Typed</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>SIMD4x2</td>
<td>Typed</td>
<td>Atomic</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>&quot;Surface Array&quot; Field in SURFACE_STATE</th>
<th>U Address</th>
<th>V Address</th>
<th>R Address</th>
<th>LOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURFTYPE_1D</td>
<td>disabled</td>
<td>X pixel address</td>
<td></td>
<td></td>
<td>LOD</td>
</tr>
<tr>
<td></td>
<td>enabled</td>
<td>X pixel address</td>
<td>array index</td>
<td></td>
<td>LOD</td>
</tr>
<tr>
<td>SURFTYPE_2D</td>
<td>disabled</td>
<td>X pixel address</td>
<td>Y pixel address</td>
<td></td>
<td>LOD</td>
</tr>
<tr>
<td></td>
<td>enabled</td>
<td>X pixel address</td>
<td>Y pixel address</td>
<td>array index</td>
<td>LOD</td>
</tr>
<tr>
<td>SURFTYPE_3D</td>
<td>disabled</td>
<td>X pixel address</td>
<td>Y pixel address</td>
<td>Z pixel address</td>
<td>LOD</td>
</tr>
<tr>
<td>SURFTYPE_BUFFER</td>
<td>disabled</td>
<td>buffer index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SURFTYPE_STRBUF</td>
<td>disabled</td>
<td>buffer index</td>
<td>byte offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** For 1D surface type SIMD4x2, the array index must be placed in the R address parameter instead of the V address parameter.
## Typed Surface Read/Write Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 13  | **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. This field is ignored if the header is not present.  
  Format = U1  
  0: Use low 8 slots  
  1: Use high 8 slots |
| 12  | Ignored |
| 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:**  
  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** |
### Typed Surface Read/Write Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 13:12 | 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. This field is ignored if the header is not present.  
Format = U2  
00: SIMD4x2  
01: Use low 8 slots  
10: Use high 8 slots  
11: 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:  
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 |
Untyped Surface ReadWrite Message Descriptor

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:12</td>
<td>SIMD Mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format = U2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: SIMD4x2 (valid for reads &amp; writes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: SIMD16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2: SIMD8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3: Reserved</td>
<td></td>
</tr>
</tbody>
</table>

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 Untyped Surface Write 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 Untyped Surface Read 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

Typed Atomic Operation Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Return Data Control</td>
</tr>
<tr>
<td></td>
<td>Specifies whether return data is sent back to the thread.</td>
</tr>
<tr>
<td></td>
<td>Format = Enable</td>
</tr>
<tr>
<td>12</td>
<td>Slot Group</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Format = U1</td>
</tr>
<tr>
<td></td>
<td>0: Use low 8 slots</td>
</tr>
<tr>
<td></td>
<td>1: Use high 8 slots</td>
</tr>
<tr>
<td>11:8</td>
<td>Atomic Operation Type</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Specifies the atomic operation to be performed.</td>
</tr>
<tr>
<td>0000:</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001:</td>
<td>AOP_AND</td>
</tr>
<tr>
<td>0010:</td>
<td>AOP_OR</td>
</tr>
<tr>
<td>0011:</td>
<td>AOP_XOR</td>
</tr>
<tr>
<td>0100:</td>
<td>AOP_MOV</td>
</tr>
<tr>
<td>0101:</td>
<td>AOP_INC</td>
</tr>
<tr>
<td>0110:</td>
<td>AOP_DEC</td>
</tr>
<tr>
<td>0111:</td>
<td>AOP_ADD</td>
</tr>
<tr>
<td>1000:</td>
<td>AOP_SUB</td>
</tr>
<tr>
<td>1001:</td>
<td>AOP_REVSUB</td>
</tr>
<tr>
<td>1010:</td>
<td>AOP_IMAX</td>
</tr>
<tr>
<td>1011:</td>
<td>AOP_IMIN</td>
</tr>
<tr>
<td>1100:</td>
<td>AOP_UMAX</td>
</tr>
<tr>
<td>1101:</td>
<td>AOP_UMIN</td>
</tr>
<tr>
<td>1110:</td>
<td>AOP_CMPWR</td>
</tr>
<tr>
<td>1111:</td>
<td>AOP_PREDEC</td>
</tr>
</tbody>
</table>

**Typed Atomic Operation SIMD4x2 Message Descriptor**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Return Data Control</td>
</tr>
<tr>
<td></td>
<td>Specifies whether return data is sent back to the thread.</td>
</tr>
<tr>
<td></td>
<td>Format = Enable</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>11:8</td>
<td>Atomic Operation Type</td>
</tr>
<tr>
<td></td>
<td>Specifies the atomic operation to be performed.</td>
</tr>
<tr>
<td>0000:</td>
<td>reserved</td>
</tr>
<tr>
<td>0001:</td>
<td>AOP_AND</td>
</tr>
<tr>
<td>0010:</td>
<td>AOP_OR</td>
</tr>
<tr>
<td>0011:</td>
<td>AOP_XOR</td>
</tr>
<tr>
<td>0100:</td>
<td>AOP_MOV</td>
</tr>
<tr>
<td>0101:</td>
<td>AOP_INC</td>
</tr>
</tbody>
</table>

---

220
### Untyped Atomic Operation Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110: AOP_DEC</td>
<td></td>
</tr>
<tr>
<td>0111: AOP_ADD</td>
<td></td>
</tr>
<tr>
<td>1000: AOP_SUB</td>
<td></td>
</tr>
<tr>
<td>1001: AOP_REVSUB</td>
<td></td>
</tr>
<tr>
<td>1010: AOP_IMAX</td>
<td></td>
</tr>
<tr>
<td>1011: AOP_IMIN</td>
<td></td>
</tr>
<tr>
<td>1100: AOP_UMAX</td>
<td></td>
</tr>
<tr>
<td>1101: AOP_UMIN</td>
<td></td>
</tr>
<tr>
<td>1110: AOP_CMPWR</td>
<td></td>
</tr>
<tr>
<td>1111: AOP_PREDEC</td>
<td></td>
</tr>
</tbody>
</table>

### Untyped Atomic Operation Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Return Data Control</td>
</tr>
<tr>
<td></td>
<td>Specifies whether return data is sent back to the thread.</td>
</tr>
<tr>
<td></td>
<td>Format = Enable</td>
</tr>
<tr>
<td>12</td>
<td>SIMD Mode</td>
</tr>
<tr>
<td></td>
<td>Format = U1</td>
</tr>
<tr>
<td>0: SIMD16</td>
<td></td>
</tr>
<tr>
<td>1: SIMD8</td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>Atomic Operation Type</td>
</tr>
<tr>
<td></td>
<td>Specifies the atomic operation to be performed.</td>
</tr>
<tr>
<td>0000: 0000: AOP_CMPWR8B</td>
<td></td>
</tr>
<tr>
<td>0001: AOP_AND</td>
<td></td>
</tr>
<tr>
<td>0010: AOP_OR</td>
<td></td>
</tr>
<tr>
<td>0011: AOP_XOR</td>
<td></td>
</tr>
<tr>
<td>0100: AOP_MOV</td>
<td></td>
</tr>
<tr>
<td>0101: AOP_INC</td>
<td></td>
</tr>
<tr>
<td>0110: AOP_DEC</td>
<td></td>
</tr>
<tr>
<td>0111: AOP_ADD</td>
<td></td>
</tr>
<tr>
<td>1000: AOP_SUB</td>
<td></td>
</tr>
</tbody>
</table>
### Untyped Atomic Operation SIMD4x2 Message Descriptor

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.

**Notes**

<table>
<thead>
<tr>
<th>Project</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not use operations with a data operand. Use only INC, DEC, or PREDEC.</td>
<td></td>
</tr>
<tr>
<td>For Atomic Counter OPS other than INC, DEC, or PREDEC, the message header is forbidden and not optional.</td>
<td></td>
</tr>
</tbody>
</table>

**Atomic Counter Operation SIMD4x2 Message Descriptor**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.

Notes

<table>
<thead>
<tr>
<th>Project</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Do not use operations with a data operand. Use only INC, DEC, or PREDEC.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:16</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Pixel/Sample Mask.</strong> 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, <strong>Slot Group</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
</tbody>
</table>
| 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.**

\[\text{Format} = \text{GeneralStateOffset}[31:10]\] |
| 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Slot 7 U Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the U Address for slot 7. Format = U32</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Slot 6 U Address</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Slot 5 U Address</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td>Slot 4 U Address</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Slot 3 U Address</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Slot 2 U Address</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Slot 1 U Address</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td>Slot 0 U Address</td>
</tr>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>Slot 15 U Address</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>Slot 14 U Address</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>Slot 13 U Address</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>Slot 12 U Address</td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td>Slot 11 U Address</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td>Slot 10 U Address</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td>Slot 9 U Address</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td>Slot 8 U Address</td>
</tr>
<tr>
<td>M3</td>
<td></td>
<td>Slots 7:0 V Address</td>
</tr>
<tr>
<td>M4</td>
<td></td>
<td>Slots 15:8 V Address</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5.7</td>
<td>31:0</td>
<td>Slot 7 Source0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations</td>
</tr>
<tr>
<td>M5.6</td>
<td>31:0</td>
<td>Slot 6 Source0</td>
</tr>
<tr>
<td>M5.5</td>
<td>31:0</td>
<td>Slot 5 Source0</td>
</tr>
<tr>
<td>M5.4</td>
<td>31:0</td>
<td>Slot 4 Source0</td>
</tr>
<tr>
<td>M5.3</td>
<td>31:0</td>
<td>Slot 3 Source0</td>
</tr>
<tr>
<td>M5.2</td>
<td>31:0</td>
<td>Slot 2 Source0</td>
</tr>
<tr>
<td>M5.1</td>
<td>31:0</td>
<td>Slot 1 Source0</td>
</tr>
<tr>
<td>M5.0</td>
<td>31:0</td>
<td>Slot 0 Source0</td>
</tr>
<tr>
<td>M6.7</td>
<td>31:0</td>
<td>Slot 15 Source0</td>
</tr>
<tr>
<td>M6.6</td>
<td>31:0</td>
<td>Slot 14 Source0</td>
</tr>
<tr>
<td>M6.5</td>
<td>31:0</td>
<td>Slot 13 Source0</td>
</tr>
<tr>
<td>M6.4</td>
<td>31:0</td>
<td>Slot 12 Source0</td>
</tr>
<tr>
<td>M6.3</td>
<td>31:0</td>
<td>Slot 11 Source0</td>
</tr>
<tr>
<td>M6.2</td>
<td>31:0</td>
<td>Slot 10 Source0</td>
</tr>
<tr>
<td>M6.1</td>
<td>31:0</td>
<td>Slot 9 Source0</td>
</tr>
<tr>
<td>M6.0</td>
<td>31:0</td>
<td>Slot 8 Source0</td>
</tr>
<tr>
<td>M7</td>
<td></td>
<td>Slots 7:0 Source1</td>
</tr>
<tr>
<td>M8</td>
<td></td>
<td>Slots 15:8 Source1</td>
</tr>
</tbody>
</table>
### SIMD16 Source Payload (AOP_CMPWR8B Only)

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

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

**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 Notes:
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.

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

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| M5.7  | 31:0| Slot 7 Red  
Specifies the value of the red channel to be written for slot 7.  
For Untyped messages:  
Format = 32 bits raw data.  
For Typed 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.

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

**SIMD4x2 Address Payload**

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Slot 1 LOD</td>
<td>HSW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Programming Note: This register can only be delivered for the <em>Typed</em> message types.</td>
<td></td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Slot 1 R Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Programming Note: This register can only be delivered for the <em>Typed</em> message types.</td>
<td></td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Slot 1 V Address</td>
<td></td>
</tr>
</tbody>
</table>
DWord | Bits | Description | Project
---|---|---|---
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>Slot 1 Source1</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>Slot 1 Source0</td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td>Slot 0 Source1</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td>Slot 0 Source0</td>
</tr>
</tbody>
</table>

**SIMD4x2 Source Payload (AOP_CMPWR8B Only)**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>Slot 1 Source1 [63:32]</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>Slot 1 Source1 [31:0]</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>Slot 1 Source0 [63:32]</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>Slot 1 Source0 [31:0]</td>
</tr>
</tbody>
</table>
### SIMD4x2 Write Data Payload (Write Message Only)

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| M2.7  | 31:0| Slot 1 Alpha  
Specifies the value of the red channel to be written for slot 7.  
For Untyped messages:  
Format = 32 bits raw data.  
For Typed 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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>DWord[Offset7]</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>DWord[Offset6]</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>DWord[Offset5]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>DWord[Offset4]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>DWord[Offset3]</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>DWord[Offset2]</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>DWord[Offset1]</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>DWord[Offset0]</td>
</tr>
</tbody>
</table>
## SIMD8 QWORD Read

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>63:0</td>
<td>QWord[Offset3]</td>
</tr>
<tr>
<td>W0.6</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W0.5</td>
<td>63:0</td>
<td>QWord[Offset2]</td>
</tr>
<tr>
<td>W0.4</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W0.3</td>
<td>63:0</td>
<td>QWord[Offset1]</td>
</tr>
<tr>
<td>W0.2</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W0.1</td>
<td>63:0</td>
<td>QWord[Offset0]</td>
</tr>
<tr>
<td>W0.0</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W1.7</td>
<td>63:0</td>
<td>QWord[Offset7]</td>
</tr>
<tr>
<td>W1.6</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W1.5</td>
<td>63:0</td>
<td>QWord[Offset6]</td>
</tr>
<tr>
<td>W1.4</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W1.3</td>
<td>63:0</td>
<td>QWord[Offset5]</td>
</tr>
<tr>
<td>W1.2</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>W1.1</td>
<td>63:0</td>
<td>QWord[Offset4]</td>
</tr>
<tr>
<td>W1.0</td>
<td>63:0</td>
<td></td>
</tr>
</tbody>
</table>
**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.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| W0.7  | 31:0| **Slot 7 Red**: Specifies the value of the red channel for slot 7.  
For Untyped messages:  
Format = 32 bits raw data.  
For Typed 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.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Slot 1 Alpha</strong>: Specifies the value of the pixel's alpha channel. Format = 32 bits raw data.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 1 Blue</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 1 Green</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 1 Red</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 0 Alpha</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 0 Blue</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 0 Green</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Red</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Slot 7 Return Data</strong>: Specifies the value of the return data for slot 7. Format = U32</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 6 Return Data</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 5 Return Data</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 4 Return Data</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 3 Return Data</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 2 Return Data</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 1 Return Data</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Return Data</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Slot 15 Return Data</td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Slot 14 Return Data</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Slot 13 Return Data</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Slot 12 Return Data</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Slot 11 Return Data</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Slot 10 Return Data</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Slot 9 Return Data</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Slot 8 Return Data</td>
</tr>
</tbody>
</table>
**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.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Slot 7 Return Data:</strong> Specifies the value of the return data for slot 7. Format = U32</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 6 Return Data</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 5 Return Data</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 4 Return Data</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 3 Return Data</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 2 Return Data</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 1 Return Data</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Return Data</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td><strong>Slot 7 Return Data[31:0]</strong>: Specifies the value of the return data for slot 7. Format = U32</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 6 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 5 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 4 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 3 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 2 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 1 Return Data[31:0]</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Return Data[31:0]</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Slot 7 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Slot 6 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Slot 5 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Slot 4 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Slot 3 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Slot 2 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Slot 1 Return Data[63:32]</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Slot 0 Return Data[63:32]</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td><strong>Slot 1 Return Data:</strong> Specifies the value of the return data for slot 1. Format = U32</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Return Data</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 1 Return Data: [63:32]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 1 Return Data: [31:0]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>reserved – not written to GRF</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 0 Return Data: [63:32]</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Return Data[31:0]</td>
</tr>
</tbody>
</table>

**Message Descriptor**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Invalidate After Read Enable</td>
</tr>
<tr>
<td></td>
<td>[DevIVB+] only</td>
</tr>
<tr>
<td></td>
<td>This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable</td>
</tr>
<tr>
<td>12:11</td>
<td>Message sub-type:</td>
</tr>
<tr>
<td></td>
<td>00: OWord Block Read/Write</td>
</tr>
<tr>
<td></td>
<td>01: Unaligned OWord Block Read</td>
</tr>
<tr>
<td></td>
<td>10: OWord Dual Block Read/Write</td>
</tr>
<tr>
<td></td>
<td>11: HWord Block Read/Write</td>
</tr>
<tr>
<td>10:8</td>
<td><strong>Block Size.</strong> Specifies the number of elements transferred see table below</td>
</tr>
</tbody>
</table>

246
## Untyped Atomic Float Add Operation Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:16</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Pixel/Sample Mask.</strong> 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.</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td><strong>Immediate Buffer Base Address.</strong> 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]</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Slot 7 U Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the U Address for slot 7. Format = U32</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Slot 6 U Address</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Slot 5 U Address</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td>Slot 4 U Address</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Slot 3 U Address</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Slot 2 U Address</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Slot 1 U Address</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td>Slot 0 U Address</td>
</tr>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>Slot 15 U Address</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>Slot 14 U Address</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>Slot 13 U Address</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>Slot 12 U Address</td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td>Slot 11 U Address</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td>Slot 10 U Address</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td>Slot 9 U Address</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td>Slot 8 U Address</td>
</tr>
<tr>
<td>M3</td>
<td></td>
<td>Slots 7:0 V Address</td>
</tr>
<tr>
<td>M4</td>
<td></td>
<td>Slots 15:8 V Address</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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    | 31:0| 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3.7</td>
<td>31:0</td>
<td>DWord[slot7]</td>
</tr>
<tr>
<td>M3.6</td>
<td>31:0</td>
<td>DWord[slot6]</td>
</tr>
<tr>
<td>M3.5</td>
<td>31:0</td>
<td>DWord[slot5]</td>
</tr>
<tr>
<td>M3.4</td>
<td>31:0</td>
<td>DWord[slot4]</td>
</tr>
<tr>
<td>M3.3</td>
<td>31:0</td>
<td>DWord[slot3]</td>
</tr>
<tr>
<td>M3.2</td>
<td>31:0</td>
<td>DWord[slot2]</td>
</tr>
<tr>
<td>M3.1</td>
<td>31:0</td>
<td>DWord[slot1]</td>
</tr>
<tr>
<td>M3.0</td>
<td>31:0</td>
<td>DWord[slot0]</td>
</tr>
<tr>
<td>M4.7</td>
<td>31:0</td>
<td><strong>DWord[slot15]</strong>. This message register is included only for SIMD16</td>
</tr>
<tr>
<td>M4.6</td>
<td>31:0</td>
<td>DWord[slot14]</td>
</tr>
<tr>
<td>M4.5</td>
<td>31:0</td>
<td>DWord[slot13]</td>
</tr>
<tr>
<td>M4.4</td>
<td>31:0</td>
<td>DWord[slot12]</td>
</tr>
<tr>
<td>M4.3</td>
<td>31:0</td>
<td>DWord[slot11]</td>
</tr>
<tr>
<td>M4.2</td>
<td>31:0</td>
<td>DWord[slot10]</td>
</tr>
<tr>
<td>M4.1</td>
<td>31:0</td>
<td>DWord[slot9]</td>
</tr>
<tr>
<td>M4.0</td>
<td>31:0</td>
<td>DWord[slot8]</td>
</tr>
</tbody>
</table>
## SIMD16/SIMD8 QWORD Source Payload

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3.7</td>
<td>63:0</td>
<td>QWord[slot3]</td>
</tr>
<tr>
<td>M3.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3.5</td>
<td>63:0</td>
<td>QWord[slot2]</td>
</tr>
<tr>
<td>M3.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3.3</td>
<td>63:0</td>
<td>QWord[slot1]</td>
</tr>
<tr>
<td>M3.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3.1</td>
<td>63:0</td>
<td>QWord[slot0]</td>
</tr>
<tr>
<td>M3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4.7</td>
<td>63:0</td>
<td>QWord[slot7]</td>
</tr>
<tr>
<td>M4.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4.5</td>
<td>63:0</td>
<td>QWord[slot6]</td>
</tr>
<tr>
<td>M4.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4.3</td>
<td>63:0</td>
<td>QWord[slot5]</td>
</tr>
<tr>
<td>M4.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4.1</td>
<td>63:0</td>
<td>QWord[slot4]</td>
</tr>
<tr>
<td>M4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M5</td>
<td></td>
<td>QWord[slot11:slot8]. This register is only included for SIMD16.</td>
</tr>
<tr>
<td>M6</td>
<td></td>
<td>QWord[slot15:slot12]. This register is only included for SIMD16.</td>
</tr>
</tbody>
</table>
**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.
### Message Descriptor

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0 = Read, 1 = Write</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><strong>Channel Mode:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: <strong>OWord</strong> – 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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: <strong>DWord</strong> – 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.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><strong>Invalidate After Read.</strong> Indicates whether the cache line should be invalidated after the read:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: No Invalidate.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: Invalidate cache line.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td><strong>Block Size.</strong> Indicates the number of SIMD8 registers to be read or written:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00: 1 register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01: 2 registers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11: 4 registers</td>
<td></td>
</tr>
<tr>
<td>11:0</td>
<td><strong>Offset.</strong> A 12-bit HWord offset into the memory Immediate Memory buffer as specified by binding table 0xFF.</td>
<td></td>
</tr>
</tbody>
</table>
### Message Header

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:16</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td><strong>Immediate Buffer Base Address.</strong> 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 <strong>General State Base Address</strong>. Format = GeneralStateOffset[31:10]</td>
</tr>
<tr>
<td></td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

### Message Payload (Write)

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7:0</td>
<td>255:0</td>
<td>HWord[Offset]</td>
</tr>
<tr>
<td>M2.7:0</td>
<td>255:0</td>
<td>HWord[Offset+1]</td>
</tr>
<tr>
<td>M3.7:0</td>
<td>255:0</td>
<td>HWord[Offset+2]</td>
</tr>
<tr>
<td>M3.7:0</td>
<td>255:0</td>
<td>HWord[Offset+3]</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:0</td>
<td>255:0</td>
<td>HWord[Offset]</td>
</tr>
<tr>
<td>W1.7:0</td>
<td>255:0</td>
<td>HWord[Offset+1]</td>
</tr>
<tr>
<td>W2.7:0</td>
<td>255:0</td>
<td>HWord[Offset+2]</td>
</tr>
<tr>
<td>W3.7:0</td>
<td>255:0</td>
<td>HWord[Offset+3]</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>The memory fence operation is not required to guarantee SLM memory access ordering between multiple threads in a thread group for the sequence of a write message, a barrier message, and then a read message. (This optimization is due to implementation details of the organization of threads in a thread group, SLM memory, data port messages, and gateway barrier messages.) Beware that the memory fence is still required for non-SLM memory ordering and observability.</td>
<td></td>
</tr>
<tr>
<td>The untyped UAV and typed UAV support are both provided by the data cache. For a thread to ensure both untyped and typed UAV are visible, the thread issues a memory fence message to the data cache data port, and the Commit Enable is no longer required. The data cache ensures that all accesses from that thread prior to the fence are visible, before any access from that thread issued after the fence will become visible. The Commit Enable is only needed if SW needs to ensure any access outside of the data cache and accesses that use the data cache are both visible before continuing. There is no known use case for this at the present time. If such a use case is needed, the thread would then insert an instruction that sources the destination registers from both memory fences before any further data port messages are sent.</td>
<td></td>
</tr>
</tbody>
</table>
Message Descriptor

<table>
<thead>
<tr>
<th>Project</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 13      |      | **Commit Enable**  
  Specifies whether the commit is returned to the thread after the fence has been honored.  
  Format = Enable |
| 12:8    |      | **Reserved: MBZ** |
| 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** |

Message Header

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7.0</td>
<td>31.0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>Message Type</th>
<th>oMask Present</th>
<th>Source Depth Present</th>
<th>Source 0 Alpha Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
</tr>
<tr>
<td>M6</td>
<td>M7</td>
<td>M8</td>
<td>M9</td>
</tr>
<tr>
<td>M10</td>
<td>M11</td>
<td>M12</td>
<td>M13</td>
</tr>
<tr>
<td>M14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Type</td>
<td>oMask Present</td>
<td>Source Depth Present</td>
<td>Source 0 Alpha Present</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**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.

General Restrictions

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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Mask.</strong> The execution mask for render target messages is ignored. Control of which pixels are active is controlled by the Pixel/Sample Enables fields in the message header.</td>
<td></td>
</tr>
<tr>
<td><strong>Execution Mask.</strong> 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.</td>
<td></td>
</tr>
</tbody>
</table>

**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 surpressed.

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Surface Format Name</th>
<th>Alpha Blend?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R32G32B32A32_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32G32_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>B8G8R8A8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B8G8R8A8_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R10G10B10A2_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Surface Format Name</td>
<td>Alpha Blend?</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>R10G10B10A2_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R8G8B8A8_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8B8A8_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8B8A8_SNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8B8A8_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R8G8B8A8_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16G16_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R16G16_SNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R16G16_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16G16_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16G16_FLOAT</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B10G10R10A2_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B10G10R10A2_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R11G11B10_FLOAT</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R32_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R32_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R32_FLOAT</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G6R5_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G6R5_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G5R5A1_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G5R5A1_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B4G4R4A4_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B4G4R4A4_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8_SNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8G8_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R8G8_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R16_SNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R16_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16(UINT)</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R16_FLOAT</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G5R5X1_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>B5G5R5X1_UNORM_SRGB</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8_SNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R8_SINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>R8_UINT</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>A8_UNORM</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Project</td>
<td>Surface Format Name</td>
<td>Alpha Blend?</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>YCRCB_NORMAL</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>YCRCB_SWAPUVY</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>YCRCB_SWAPUV</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>YCRCB_SWAPY</td>
<td>No</td>
</tr>
</tbody>
</table>

**Subspan/Pixel 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. However NumSamples = 4X is supported only on [DevGT+]. NumSamples = 8X is supported only on [HSW].

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)

<table>
<thead>
<tr>
<th>Dispatch Size</th>
<th>Num Samples</th>
<th>Slot Mapping (SSPI = Starting Sample Pair Index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD32</td>
<td>1X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td>Dispatch Size</td>
<td>Num Samples</td>
<td>Slot Mapping (SSPI = Starting Sample Pair Index)</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>4X</td>
<td></td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td>SIMD16</td>
<td>8X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>1X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
</tbody>
</table>

Restriction:
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**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:8</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Dispatch ID.</strong> 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.</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td><strong>Pixel Mask.</strong> 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.</td>
</tr>
</tbody>
</table>

The bits in this mask correspond to the pixels as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>16</th>
<th>17</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
### DWord, Bit, Description

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>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</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>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</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Slot 7 Red. Specifies the value of the red channel for slot 7. Format = 32 bits raw data.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 6 Red</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 5 Red</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 4 Red</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 3 Red</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 2 Red</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 1 Red</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Red</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Slot 15 Red</td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Slot 14 Red</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Slot 13 Red</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Slot 12 Red</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Slot 11 Red</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Slot 10 Red</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Slot 9 Red</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Slot 8 Red</td>
</tr>
<tr>
<td>W2</td>
<td></td>
<td>Slots 7:0 Green</td>
</tr>
<tr>
<td>W3</td>
<td></td>
<td>Slots 15:8 Green</td>
</tr>
<tr>
<td>W4</td>
<td></td>
<td>Slots 7:0 Blue</td>
</tr>
<tr>
<td>W5</td>
<td></td>
<td>Slots 15:8 Blue</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td></td>
<td>Slots 7:0 Alpha</td>
</tr>
<tr>
<td>W7</td>
<td></td>
<td>Slots 15:8 Alpha</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Slot 7 Red. Specifies the value of the red channel for slot 7. Format = 32 bits raw data.</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Slot 6 Red</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Slot 5 Red</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Slot 4 Red</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Slot 3 Red</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Slot 2 Red</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Slot 1 Red</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Slot 0 Red</td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td>Slots 7:0 Green</td>
</tr>
<tr>
<td>W2</td>
<td></td>
<td>Slots 7:0 Blue</td>
</tr>
<tr>
<td>W3</td>
<td></td>
<td>Slots 7:0 Alpha</td>
</tr>
</tbody>
</table>
### Header for SIMD8_IMAGE_WRITE

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:10</td>
<td>Ignored</td>
</tr>
<tr>
<td>9:8</td>
<td></td>
<td><strong>Color Code:</strong> This ID is assigned by the Windower unit and is used to track synchronizing events. Format: Reserved for HW Implementation Use.</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>FFTID.</strong> 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.</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:3</td>
<td>Ignored</td>
</tr>
<tr>
<td>2:0</td>
<td></td>
<td><strong>Render Target Index.</strong> Specifies the render target index that will be used to select blend state from BLEND_STATE. Format = U3</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:6</td>
<td><strong>ColorCalculatorState Pointer.</strong> 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, SW is recommended to program a dummy color-calc state such that all operations controlled by this state are disabled.</td>
</tr>
<tr>
<td>5:0</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:27</td>
<td>Ignored</td>
</tr>
<tr>
<td>30:27</td>
<td></td>
<td><strong>Viewport Index.</strong> 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.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|       | 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>
<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 **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 |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>X5: x-coordinate for pixel 5 Format = U16</td>
<td></td>
</tr>
<tr>
<td>M1.4</td>
<td>31:16</td>
<td>Y4: y-coordinate for pixel 4 Format = U16</td>
</tr>
<tr>
<td>15:0</td>
<td>X4: x-coordinate for pixel 4 Format = U16</td>
<td></td>
</tr>
<tr>
<td>M1.3</td>
<td>31:16</td>
<td>Y3: y-coordinate for pixel 3 Format = U16</td>
</tr>
<tr>
<td>15:0</td>
<td>X3: x-coordinate for pixel 3 Format = U16</td>
<td></td>
</tr>
<tr>
<td>M1.2</td>
<td>31:16</td>
<td>Y2: y-coordinate for pixel 2 Format = U16</td>
</tr>
<tr>
<td>15:0</td>
<td>X2: x-coordinate for pixel 2 Format = U16</td>
<td></td>
</tr>
<tr>
<td>M1.1</td>
<td>31:16</td>
<td>Y1: y-coordinate for pixel 1 Format = U16</td>
</tr>
<tr>
<td>15:0</td>
<td>X1: x-coordinate for pixel 1 Format = U16</td>
<td></td>
</tr>
<tr>
<td>M1.0</td>
<td>31:16</td>
<td>Y0: y-coordinate for pixel 0 Format = U16</td>
</tr>
<tr>
<td>15:0</td>
<td>X0: x-coordinate for pixel 0 Format = U16</td>
<td></td>
</tr>
</tbody>
</table>
**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.

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

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

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

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

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

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.

The dual source message contains only 2 subspans (8 pixels) due to limitations in message length.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mm.7</td>
<td>31:0</td>
<td><strong>Slot 7 Source 0 Red.</strong> Specifies the value of the slot’s red component. Format = IEEE Float, S31, or U32 depending on the <strong>Surface Format</strong> of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.</td>
</tr>
<tr>
<td>Mm.6</td>
<td>31:0</td>
<td>Slot 6 Source 0 Red</td>
</tr>
<tr>
<td>Mm.5</td>
<td>31:0</td>
<td>Slot 5 Source 0 Red</td>
</tr>
<tr>
<td>Mm.4</td>
<td>31:0</td>
<td>Slot 4 Source 0 Red</td>
</tr>
<tr>
<td>Mm.3</td>
<td>31:0</td>
<td>Slot 3 Source 0 Red</td>
</tr>
<tr>
<td>Mm.2</td>
<td>31:0</td>
<td>Slot 2 Source 0 Red</td>
</tr>
<tr>
<td>Mm.1</td>
<td>31:0</td>
<td>Slot 1 Source 0 Red</td>
</tr>
<tr>
<td>Mm.0</td>
<td>31:0</td>
<td>Slot 0 Source 0 Red</td>
</tr>
<tr>
<td>M(m+1)</td>
<td>31:0</td>
<td><strong>Slot[7:0] Source 0 Green.</strong> See Mm definition for slot locations.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>M(m+2)</td>
<td></td>
<td>Slot[7:0] Source 0 Blue. See Mm definition for slot locations.</td>
</tr>
<tr>
<td>M(m+3)</td>
<td></td>
<td>Slot[7:0] Source 0 Alpha. See Mm definition for slot locations.</td>
</tr>
<tr>
<td>M(m+4)</td>
<td></td>
<td>Slot[7:0] Source 1 Red. See Mm definition for slot locations.</td>
</tr>
<tr>
<td>M(m+5)</td>
<td></td>
<td>Slot[7:0] Source 1 Green. See Mm definition for slot locations.</td>
</tr>
<tr>
<td>M(m+6)</td>
<td></td>
<td>Slot[7:0] Source 1 Blue. See Mm definition for slot locations.</td>
</tr>
<tr>
<td>M(m+7)</td>
<td></td>
<td>Slot[7:0] Source 1 Alpha. See Mm definition for slot locations.</td>
</tr>
</tbody>
</table>
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:
- $s_0, s_1 =$ source 0, source 1
- $1/0 =$ slots 15:8
- $3/2 =$ slots 7:0
- $s_Z =$ source depth
- $oM =$ oMask

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

278
### 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.

#### General Restrictions

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).

<table>
<thead>
<tr>
<th>Message Type</th>
<th>oMask Present</th>
<th>Source Depth Present</th>
<th>Source 0 Alpha Present</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>M9</th>
<th>M10</th>
<th>M11</th>
<th>M12</th>
<th>M13</th>
<th>M14</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3/2s0R</td>
<td>3/2s0G</td>
<td>3/2s0B</td>
<td>3/2s1R</td>
<td>3/2s1G</td>
<td>3/2s1B</td>
<td>3/2s1A</td>
<td>3/2s1Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>oM</td>
<td>3/2s0R</td>
<td>3/2s0G</td>
<td>3/2s0B</td>
<td>3/2s1R</td>
<td>3/2s1G</td>
<td>3/2s1B</td>
<td>3/2s1A</td>
<td>3/2s1Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>oM</td>
<td>3/2s0R</td>
<td>3/2s0G</td>
<td>3/2s0B</td>
<td>3/2s1R</td>
<td>3/2s1G</td>
<td>3/2s1B</td>
<td>3/2s1A</td>
<td>3/2s1Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>s0A</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td>sZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>s0A</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td>sZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>oM</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>oM</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>oM</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td>sZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>s0A</td>
<td>oM</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td>sZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Mask.</strong></td>
<td>The execution mask for render target messages is ignored. Control of which pixels are active is controlled by the Pixel/Sample Enables fields in the message header.</td>
</tr>
<tr>
<td><strong>Execution Mask.</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Surface Format Name</th>
<th>Alpha Blend?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R32G32B32A32_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32G32B32A32_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16B16A16_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32G32_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32G32_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>B8G8R8A8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B8G8R8A8_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R10G10B10A2_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R10G10B10A2_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Surface Format Name</td>
<td>Alpha Blend?</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>R8G8B8A8_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R8G8B8A8_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16G16_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16G16_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B10G10R10A2_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B10G10R10A2_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R11G11B10_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R32_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R32_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G6R5_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G6R5_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G5R5A1_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G5R5A1_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B4G4R4A4_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B4G4R4A4_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8G8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8G8_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8G8_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R8G8_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R16_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R16_FLOAT</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G5R5X1_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>B5G5R5X1_UNORM_SRGB</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8_SNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>R8_SINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>R8_UINT</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>A8_UNORM</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>YCRCB_NORMAL</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>YCRCB_SWAPUVY</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Surface Format Name</td>
<td>Alpha Blend?</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>YCRCB_SWAPUV</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>YCRCB_SWAPY</td>
<td>No</td>
</tr>
</tbody>
</table>

**Message Header**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:8</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Dispatch ID.</strong> 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.</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td><strong>Pixel Mask.</strong> 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. The bits in this mask correspond to the pixels as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 4 5 16 17 20 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 3 6 7 18 19 22 23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 9 12 13 24 25 28 29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 11 14 15 26 27 30 31</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td><strong>Y offset.</strong> 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</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td><strong>X offset.</strong> 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</td>
</tr>
</tbody>
</table>
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.

Skin Tone Detection/Enhancement (STD/E)

The STD/E unit, composed of the Skin Tone Detection (STD) and Skin Tone Enhancement (STE) units, is part of color processing pipe located at the Render Cache Pixel Backend (RCBP).

The main goal of the STD/E is to reproduce the skin colors in a way that is more palatable to the observer, and by that to increase the sensed image quality. It may also pass indication of skin tones to the TCC and ACE.

The STD unit detects the skin like colors and passes a grade of skin tone color to the STE. The STE modify the saturation and Hue of the pixel. Both the STD and STE are per-pixel basis. The input pixels are required to be on the YUV space.

The skin tone detected factor will be recorded as a 5-bit number and it will be passed to the module of ACE and TCC to indicate the strength of skin tone likelihood.

Input and output pixel is 444 format and 12 bits per channel. Precision=12; Prec_shift = 4

STD

Skin Tone Detection (STD) module is turned on with state parameter STD Enable. The output of STD can be either the original pixel values or the skin tone likelihood score (STD score) for every pixel, and the selection of the output type is controlled by Output Control.

The distribution of skin color can be viewed statistically as a cluster spanning in the 3D YUV domain where the center of the cluster is considered as the 'exact' skin tone value. The detection of such 3D cluster can be done by taking the intersection of its projections onto three planes (that is, UV, YV, and YU).

While the distribution of skin color can roughly result in an ellipse shape from projection on UV plane, a rectangle and a diamond shape in a transformed coordinate (i.e., saturation hue (SH) domain) are utilized by STD block to approximate the shape of the projection to facilitate the detection of the projection in UV plane. The likelihood of a pixel being a skin color pixel in UV plane, likelihood_uv is then calculated from a function modeling the distance relationship between an input pixel and the transformed/projected skin tone center in the enclosure of the rectangle and the diamond shape. Several state parameters are involved in the derivation of likelihood_uv:

- \( \cos(\alpha), \sin(\alpha) \): parameters for UV to SH transformation
- \( U_{Mid}, V_{Mid} \): configurable UV values of the pixel considered to be the skin tone center; center of the rectangle
• **Hue_Max, Sat_Max**: parameters specifying the coverage of the rectangle (thus, the range of SH values to be considered possibly as skin tone color)

• **HS_margin**: parameter indicating the transition rate of skin tone likelihood score as a pixel moves towards the center of the rectangle

• **Diamond_du, Diamond_dv**: shift amount between the center of the diamond shape and that of the rectangle

• **Diamond_alpha, Diamond_Th**: parameters specifying the size and the shape of the diamond

• **Diamond Margin**: parameter indicating the transition rate of skin tone likelihood score as a pixel moves towards the center of the diamond

The projection of the 3D skin color cluster on YU is further simplified to be a 1D projection over Y axis where the skin likelihood score likelihood_yu is derived from one Piece-Wise-Linear-Function (PWLF) mapping input pixel's Y value to the likelihood score. Four control points (Y_point_1, Y_point_2, Y_point_3, Y_point_4) together with two slopes (Y_Slope_1, Y_Slope_2) are available to SW for characterizing this PWLF. While (Y_point_1, Y_point_2, Y_point_3, Y_point_4) would specify the range of Y values considered valid for skin tone color, (Y_Slope_1, Y_Slope_2) would be the transition rate of the skin likelihood score as the change of a pixel's Y value within the valid range.

The projection of the 3D skin color cluster on YV plan is approximated be covered by an enclosure of two PWLFs (i.e., upper PWLF and lower PWLF). The calculation of likelihood_yv is similar to the calculation of likelihood_uv: the likelihood would be zero for pixels fall outside the enclosure of the two PWLFs while the likelihood value increases from zero to one (in normalized scale) as the pixels inside the enclosure moves towards the center of the enclosure. External state parameters involved in the derivation of likelihood_yv are:

• (P0L, B0L), (P1L, B1L), (P2L, B2L), (P3L, B3L): control points for lower PWLF

• S0L, S1L, S2L, S3L: slopes for lower PWLF

• (P0U, B0U), (P1U, B1U), (P2U, B2U), (P3U, B3U): control points for upper PWLF

• S0U, S1U, S2U, S3U: slopes for upper PWLF

• **INV_Margin_VYL**: parameter indicating the transition rate of skin tone likelihood score as a pixel within the enclosure moves towards the center of the enclosure viewed from the lower PWLF point of view

• **INV_Margin_VYU**: parameter indicating the transition rate of skin tone likelihood score as a pixel within the enclosure moves towards the center of the enclosure viewed from the upper PWLF point of view

The final skin tone likelihood score, STD score, is taken as an intersection of the likelihood score in UV, YU, and YV plane. That is

$$\text{STD_score} = \min(\text{likelihood_uv}, \text{likelihood_yu}, \text{likelihood_yv})$$

STD score is represented as a 5bps integer number. Note that the detection of skin tone color in YV plane is controlled by the state parameter **VY_STD_Enable**, and it is optional. When the detection in YV plane is turned off, STD score is taken as the intersection of likelihood_uv and likelihood_yu.
**Skin Tone Enhancement**

The Skin Tone Enhancement step is performed on the pixels that are detected as the skin-tone pixels by the Skin Tone Detection (STD) step. The Skin Tone Enhancement step is divided into two sub-steps: Saturation Correction Enhancement and Hue Correction Enhancement.

**Saturation Correction Enhancement**

The saturation correction enhancement is performed by the transformation \( \text{Sat}_{\text{new}} = F_{\text{sat}}(\text{Sat}_{\text{old}}) \), which is realized by the piece-wise linear function \( F_{\text{sat}} \) with 4 linear segments.

The parameters of this PWLF are:

- **Points:**
  - \( \text{SATP}_0 = -\text{SatMax} \)
  - \( \text{SATP}_x \) (\( x=1,2,3 \)) – defined by the user
  - \( \text{SATP}_4 = \text{SatMax} \)

- **Biases:**
  - \( \text{SATB}_0 = -\text{SatMax} \)
  - \( \text{SATB}_x \) (\( x=1,2,3 \)) – defined by the user
  - \( \text{SATB}_4 = \text{SatMax} \)

- **Slopes:**
  - \( \text{SATS}_x \) (\( x=0,1,2,3 \)) – defined by the user

There are Programming Restrictions to specify the parameters. (See the figure below for an example.)

- The point Sat = -SatMax maps to itself: \((-\text{SatMax}) \rightarrow (-\text{SatMax})\).
- The point Sat = SatMax maps to itself: \((\text{SatMax}) \rightarrow (\text{SatMax})\).
- The correction function is continuous.
- The correction function is non-decreasing.
General Form of the Saturation Correction PWLF

**Note:** Although the points with \( Sat < -SatMax \) are processed, these points are not affected by the skin tone saturation correction enhancement, because for these points, the SkinToneFactor = 0.

**Hue Correction Enhancement**

The hue correction enhancement is performed by the transformation \( Hue_{new} = F_{Hue}(Hue_{old}) \), which is realized by the piece-wise linear function \( F_{Hue} \) with 4 linear segments.

The parameters of this PWLF are:

- **Points:**
  - \( HUEP0 = -HueMax \)
  - \( HUEPx (x=1,2,3) \) – defined by the user
  - \( HUEP4 = HueMax \)

- **Biases:**
  - \( HUEB0 = -HueMax \)
  - \( HUEBx (x=1,2,3) \) – defined by the user
  - \( HUEB4 = HueMax \)

- **Slopes:**
The point $\text{Hue} = -\text{Hue}_{\text{Max}}$ maps to itself: $(-\text{Hue}_{\text{Max}}) \rightarrow (-\text{Hue}_{\text{Max}})$.

The point $\text{Hue} = \text{Hue}_{\text{Max}}$ maps to itself: $(\text{Hue}_{\text{Max}}) \rightarrow (\text{Hue}_{\text{Max}})$.

The correction function is continuous.

The correction function is non-decreasing.

General form of the Hue Correction PWLF

Skin Type Correction Enhancement

This optional enhancement operation is enabled by setting the control parameter $\text{Skin\_Types\_Enable}$ to 1.

In this advanced optional mode, skin tone is enhanced based on one of two skin types, Bright Skin and Dark Skin. A second set of the Saturation and Hue Correction enhancement parameters is defined, with different values in an identical structure. The second set of parameters has the suffix "_DARK" in the names.

The classification of the skin type is achieved using two parameters, $\text{Skin\_types\_thresh}$, and $\text{Skin\_types\_margin}$ in the luma (Y) values.
**Bright/Dark Skin Type Classifier**

The related parameters are:

- **Points:**
  - $HUEP_{x, DARK} (x=1,2,3)$ – defined by the user
  - $SATP_{x, DARK} (x=1,2,3)$ – defined by the user

- **Biases:**
  - $HUEB_{x, DARK} (x=1,2,3)$ – defined by the user
  - $SATB_{x, DARK} (x=1,2,3)$ – defined by the user

- **Slopes:**
  - $HUES_{x, DARK} (x=0,1,2,3)$ – defined by the user
  - $SATS_{x, DARK} (x=0,1,2,3)$ – defined by the user

The final values of the skin tone enhanced pixels are given by:

$$Sat_{new} = M_{V_{dark}} * Sat_{newD} + M_{V_{bright}} * Sat_{newB}$$

$$Hue_{new} = M_{V_{dark}} * Hue_{newD} + M_{V_{bright}} * Hue_{newB}$$

where $M_{V_{dark}}$ and $M_{V_{bright}}$ are blending factors computed from the plot above.
Transformation from (Sat, Hue) to (U, V)

The (U,V) → (Sat,Hue) transformation is processed by the two steps: shift and rotation. The inverse transformation, (Sat,Hue) → is done by: a rotation and then a shift.

The difference (DU, DV) between the skin tone enhanced and original UV values is weighted by the five-bit-per-pixel skin tone detection result SkinToneFactor.

The final output (U_out,V_out) values are calculated by adding the weighted difference values (DU, DV):

\[
U_{\text{out}} = U_{\text{in}} + DU \\
V_{\text{out}} = V_{\text{in}} + DV
\]

STD Score Output

This mode outputs the STD score, which is controlled by the state bit Output STD Decisions instead of the pixel values. In this mode, the STD should be enabled and other functions in the IECP after STDE in the pipe should be disabled. Only ACE can be enabled to collect the histogram of the STD score values.

The output when Output STD Decision is enabled should be as follows:

\[
Y = 0x7FF + + (\text{STD\_Score} <<6) \\
U = 0x7FF \\
V = 0x7FF
\]

In this mode, a histogram of skin tone distribution can be obtained in ACE module, and a special ACE PWLF curve (step function) can be configurated to produce a bi-level picture to illustrate the pixels based on the level of skin tone detection.

Adaptive Contrast Enhancement (ACE)

Automatic Contrast Enhancement (ACE) is a part of the color processing pipe. It works in YCbCr444 12bpc color space.

The main goals of ACE are to improve the overall contrast of the image and to emphasize details in obscured regions, such as dark regions of the input image.

The ACE algorithm analyzes the input image and modifies contrast of the image according to its content characteristic. Analysis and contrast adjustment are performed over the Y component.

ACE algorithm generates a Piece-wise Linear Function (PWLF) that maps input luma values to output luma values. The output luma values, Yout, are calculated by \( Y_{\text{out}} = \text{PWLF}(Y_{\text{in}}) \).

ACE receives skin-tone information from the Skin Tone Enhancement block. When the input image contains skin-tone colors, the effect of ACE is reduced in the regions with skin-tone colors.

ACE works on the upper 8bit MSB of Y-ch for point detection in the PWLF.
The parameter, skin_threshold, is used to determine if the current pixel contains skin-tone color or not. The Full_image_histogram flag forces the ACE operations on all of the input pixels.

HW computes the luma histogram and the maximum and minimum luma values (Ymax, Ymin) from the input image. The number of skin-tone pixels is also computed.

The PWLF is an eleven-segment (12 points) 1D LUT and specified by the set of parameters (Points: Ymin, Y1-Y10, Ymax, Bias: B1 – B10, Slope: S0-S10).

**Total Color Control (TCC)**

TCC adjusts the color saturation level of the input image based on six anchor colors (Red, Green, Blue, Magenta, Yellow, and Cyan). The TCC algorithm operates on the UV-color components in the YUV color space on a per-pixel basis.

Input and output pixels are in the YUV444 12bpc format. The input to the TCC block is:

- U and V color components (10 bit)
- Skin-tone detection value (5 bit)
- External control parameters

The output of the TCC block is:

- Updated U and V values (10 bit)

The TCC block is implemented in HW to reduce the power of the system and improve the battery life. The throughput is two pixels per clock. See the diagram below. There are two paths in parallel to support the requirement of two pixels per clock.
The TCC block is controlled by state only and does not require any memory access. The TCC block runs at the same frequency as the existing RCPBunit.

The TCC block includes three sub-blocks: Angle_Calculator, Saturation_Factor_Calculator, UVModification.

**Angle_Calculator**

This sub-block computes the color hue angle, $\theta$, in radians (10 bit approximation with maximal error of 0.005 rad).

**Saturation_Factor_Calculator**

This sub-block uses the angle $\theta$ to find the corresponding anchor colors and calculates the multiplicative saturation factor in 8-bit per pixel.

This block requires several external input parameters such as:

- Basic Colors: $C_1, \ldots, C_6$
- Basic Saturation Factors (SFs): $SF_1, \ldots, SF_6$
- Color Transition Slopes: $\alpha_1, \ldots, \alpha_6$
- Color Biases: $b_1, \ldots, b_6$
- $UV_{thr}$, $UV_{thrBts}$
- $STE_{thr}$, $STE_{slopeBts}$
- BaseColor1, ..., BaseColor6 – Six basic user-defined colors (anchor colors)
- SatFactor1, ..., SatFactor6 – Six user-defined saturation factors for anchor colors
- ColorTransitSlope12, ..., ColorTransit61 – Six calculation results of 1/(BaseColorX – BaseColorY) for anchor colors
- ColorBias1, ..., ColorBias6 – Six color biases for anchor colors
- STDscore – Skin-Tone Detection score (from the STD/E block)

There are four intermediate saturation factors, SFs1, SFs2, SFs3, and SFs4. The final saturation factor SFFinal is equal to SFs4.

The first saturation factor SFs1 is computed from the external input parameters (SatFactori, BaseColori, ColorTransitSlopei, ColorBiasi) and the color hue angle θ.

Computation of the saturation factor SFs2 involves (UVMaxColor, Inv_UVMaxColor) where UVMaxColor is the maximum (and legal) absolute UV values, which in the case of YUV color space equals 448 in 10-bit representation. Inv_UVMaxColor is the inverse calculation of UVMaxColor, that is, 1/UVMaxColor.

The third saturation factor SFs3 involves CLF which is Color Limiting Factor and ranges from 0 to 1. CLF is computed using a threshold value UV_Threshold.

The last and forth saturation factor SFs4 considers the skin-tone pixels and a threshold value STE_Threshold.

UV Modification

The input UV pixels are multiplied by the saturation factor SFFinal in this sub-block.

The calculation of the modified output Unew and Vnew values are:

\[
\begin{align*}
U_{\text{new}} &= U \times \text{SFFinal} \\
V_{\text{new}} &= V \times \text{SFFinal}
\end{align*}
\]

where (U, V) are the input color components.

ProcAmp

The PROCAMP block modifies the brightness, contrast, hue, and saturation of the input image in YUV color space.

Input and output pixels are in the YCbCr 444 12bpc (bits per channel) format. Precision=12.
**Y Processing:**

An offset of 256 (that is, 16 in 8bpc) is subtracted from the 12-bit Y values to position the black level at zero. This removes the DC offset so that adjusting the contrast does not vary the black level. Since Y values may be less than 256, negative Y values should be supported at this point. Contrast is adjusted by multiplying the YUV pixel values by a constant. If U and V are adjusted, a color shift results whenever the contrast is changed. The brightness property value is added (or subtracted) from the contrast adjusted Y values; this is done to avoid introducing a DC offset due to adjusting the contrast. Finally the offset 256 is added back to reposition the black level at 256.

The equation for processing Y values is:

\[ Y_{out}' = ((Y_{in} - 256) \times C) + B + 256, \]

Where C is the Contrast adjustment value and B is the Brightness adjustment value.

**UV Processing:**

An offset of 2048 (that is, 128 in 8bpc) is subtracted from the 12-bit U and V values. The hue adjustment is implemented by combining the U and V input values together as in:

\[
U_{out}' = (U_{in} - 2048) \times \cos(H) + (V_{in} - 2048) \times \sin(H) \\
V_{out}' = (V_{in} - 2048) \times \cos(H) - (U_{in} - 2048) \times \sin(H)
\]

Where H represents the desired Hue angle; Saturation is adjusted by multiplying the U and V input values by a constant S.

Finally, the offset value 2048 is added back to both U and V.

The combined processing of Hue, Saturation, and Contrast on the UV data is:

\[
U_{out}' = (((U_{in} - 2048) \times \cos(H) + (V_{in} - 2048) \times \sin(H)) \times C \times S) + 2048 \\
V_{out}' = (((V_{in} - 2048) \times \cos(H) - (U_{in} - 2048) \times \sin(H)) \times C \times S) + 2048
\]

Where C is the contrast, H is Hue angle, and S is the Saturation.
The multiplication factors \(\cos(H)\times Cx\times S\) and \(\sin(H)\times Cx\times S\) are programmed by the parameters \(\cos_c_s\) and \(\sin_c_s\).

**Color Space Conversion**

The CSC block enables linear conversion between different color spaces such as YCbCr and RGB using vector shifts and matrix multiplication.

The CSC algorithm is a linear coordinate transformation, comprising of the following steps:

1. Shift the input color coordinate
2. Multiply by 3x3 matrix
3. Shift the output color coordinate

The formula representation of the 3 steps is:

\[
\begin{pmatrix}
 v_{out_1} \\
 v_{out_2} \\
 v_{out_3}
\end{pmatrix} = \begin{pmatrix}
 a_{11} & a_{12} & a_{13} \\
 a_{21} & a_{22} & a_{23} \\
 a_{31} & a_{32} & a_{33}
\end{pmatrix} \begin{pmatrix}
 v_{in_1} + v_{0_1} \\
 v_{in_2} + v_{0_2} \\
 v_{in_3} + v_{0_3}
\end{pmatrix} + \begin{pmatrix}
 u_{0_1} \\
 u_{0_2} \\
 u_{0_3}
\end{pmatrix}
\]

where

- \(a_{ij}\) are the 3x3 matrix elements \([C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8]\) in S2.10
- \(v_{in_i}\) are the color components of the input pixel in U12
- \(v_{out_i}\) are the color components of the output pixel in U12
- \(v_{0_i}\) are the input offset vector elements \([Offset\_in\_1, Offset\_in\_2, Offset\_in\_3]\) in S10
- \(u_{0_i}\) are the output offset vector elements \([Offset\_out\_1, Offset\_out\_2, Offset\_out\_3]\) in S10

The output pixel values are clipped to ensure that each color component is within the valid range.
Color Gamut Compression

Background of Color Gamut Compression

While most photography today complies with the sRGB standard color space, which covers around 72% of the color perceived by humans, this 72% content looks incorrect/unnatural on wide gamut displays, which can extend more than 100%. Therefore, a gamut mapping (GM) algorithm is required to adjust when the input gamut range is different from the output gamut range such as an input sRGB color space displayed on a wide gamut display, or to adjust wide gamut content to display on traditional lower gamut displays.

The easiest compression method applied to displaying wider gamut content on lower gamut displays is to clip the out of range primary values to the valid range (i.e., 0-1). Although this simple clipping procedure leads to acceptable visual appearance in most cases, loss of color depth can be observed in the video containing out-of-range pixels. The reason behind this effect should be the uniform quantization process applied to out-of-range values (e.g., two distinct out-of-range red colors are mapped to the same boundary red color). Moreover, the simple clipping method treats each color channel independently. This may lead to unexpected perceptual loss since the composite ratios of three primaries have been changed. An approach which takes these two factors into account while scaling down the out of range values can possibly maintain the detail information of the image.

<table>
<thead>
<tr>
<th>Project:</th>
<th>SNB+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input and output pixel is 444 format and 12bits per channel. Precision=12; Prec_shift = 4.</td>
<td></td>
</tr>
</tbody>
</table>

Usage Models

There are two usage models depending on the set up of the `FullRangeMappingEnable` bit:

Basic mode: fixed-hue color gamut clipping mode

Advanced mode: fixed-hue full range mapping mode

The application of basic mode of the fixed-hue color gamut clipping is preferred when the content having the smaller percentage of out-of-range pixels in the scene. The advanced mode of fixed-hue full range mapping mode may also change the in-range pixels and is thus preferred when the percentage of out-of-range pixel is large. The outcome of the in/out range pixel percentage is derived from the out-of range color gamut detection module to provide an indicator to operate among basic mode and advanced mode.
Gamut Compression Module Overview

The main goal of color gamut compression module algorithm is to compress out-of-range pixel values while keeping their hue values same as it is before compression. A block diagram to color gamut compress the xv Color video into sRGB format is shown below.

AT the pipeline level, the input into Gamut compression unit is from STDE unit and the output from the Gamut compression goes to TCCE unit. The Gamut compression comprises of the following stages:

xvYCC decoding

YUV2LCH color space conversion

Out of range Gamut pixel detection

Scaling factor calculation

Find out the Euclidean distance for the out of range pixel for advance mode

Fixed-hue Gamut compression

Bring the out of range pixel to the boundary for basic mode

Bring the out of range pixel depending on the distance and apply uniform quantization process in advance mode

xvYCC encoding

**xvYCC decoding**

The input non-linear YCbCr values (i.e., Y’Cb’Cr’, or Y’UV) are decoded into linear, normalized yuv space.
Depending on the characteristics (e.g., linearity) of the input to gamut compression module, the state parameter `xvYCCDecEncEnable` can be utilized to indicate whether xvYCC decoding (later with xvYCC encoding) needs to be turned on.

`Src prec` is 12.

**YUV2LCH**

The scaling factors utilized to compress the out-of-gamut pixels are derived in Lightness-Chroma-Hue (LCH) color-space. The conversion from linear yuv space to LCH space is known as:

\[
\begin{align*}
    l &= y \\
    c &= \sqrt{u^2 + v^2} \\
    h &= \tan^{-1}(v/u)
\end{align*}
\]

**Out-of-Gamut Pixel Detection**

For the mapping of xvYCC to sRGB, both gamuts share the same primaries as those defined in either BT.601 or BT.709. With the transformation matrix available for yuv-rgb conversion in BT.601 or BT.709, the sRGB gamut boundary can be depicted as a rectangle for each hue plane. The rectangle which is the sRGB gamut boundary for a hue is characterized by its three vertices: (0, 0), (Cv_h, Lv_h), and (0, 1). An example of the concept of gamut boundary is given in below figure.
Gamut boundary: the sRGB gamut boundary is a rectangle shape for each hue plane. p1, p2, p3, p4 are pixels with the same hue value.

**VEBOX_VERTEX_TABLE** serves as the storage place for the programmed vertex information \((Cv_h, Lv_h)\) where the continuous \([0, \pi)\) hue plane is quantized to be 512 resulting in a 512x2-entries table with 12bpc for every entry.

With the destination gamut depicted as aggregation of rectangles over all hue planes, out-of-gamut pixel detection block works in LC-plane indexed by a pixel's hue value and determine whether the pixel falls outside the rectangle. For example, p1, p2, and p3 in the above figure would be detected as out-of-gamut pixels. A statistic parameter, **number_of_out-of-range_pixel** collects the number of out-of-gamut pixels at picture level through VSC unit. This statistic parameter may be utilized to assess the property of a picture to achieve intelligent selection of gamut compression mode applied to the input picture.

**Scaling Factor – Basic Mode**

The slope of a compression line is defined from the vertex point table.

\[
m_{\text{comp}} = m_{\text{vert}} \gg (\text{compression\_line\_shift}), \text{ with } \text{compression\_line\_shift} \text{ default to be } 3.(5)
\]
\( m_{\text{comp}} \) in the above equation is the slope of the compression line while \( m_{\text{vert}} \) represents the slope of the line perpendicular to the RGB boundary line:

\[
m_{\text{vert}} = -\frac{1}{m_{\text{boundary}}}, \quad \text{and (6)}
\]

\[
m_{\text{boundary}} = \frac{(l_{V} - e_{V})}{c_{V}}
\]

where

\[
e_{V} = \begin{cases} 1, & \text{if } l_{p_i} > l_{V} \\ 0, & \text{else} \end{cases}
\]

The intersection between the compression line for pixel \( p_i \) and the \( L \)-axis is denoted as \( I_{P_i} \).

\[
I_{P_i} = (c_{I_{P_i}}, l_{I_{P_i}})
\]

then

\[
c_{I_{P_i}} = 0, \quad \text{and (8)}
\]

\[
l_{I_{P_i}} = l_{P_i} - c_{P_i} \times m_{\text{comp}}
\]

The point nearest to the input pixel \( p_i \) on the RGB boundary along the compression direction (i.e., intersection between the compression line and the RGB boundary) be \( B_{P_i} \), then

\[
B_{P_i} = (c_{B_{P_i}}, l_{B_{P_i}})
\]

with

\[
c_{B_{P_i}} = \frac{(l_{P_i} - e_{V})}{(m_{\text{boundary}} - m_{\text{comp}})}, \quad \text{and (9)}
\]

\[
l_{B_{P_i}} = c_{B_{P_i}} \times m_{\text{boundary}} + e_{V}
\]

Scaling factor is denoted as
\[ s f_{p_i} = \frac{c_{p_i, output}}{c_{p_i}} \]  

(10)

For the usage of Basic mode - fixed-hue color gamut clipping mode, all out-of-range pixels will be clipped to the boundary, which means

\[ c_{p_i, output} = c_B p_i \]  

(11)

And the luma is mapped at along the compression line to hit the boundary line at

\[ l_{p_i, output} = l_{p_i} + c_{p_i, output} \times m_{comp} \]  

(12)

**Fixed-Hue Compression**

The output of fixed compression is based on the scaling factor and the property of pixel.

\[
\begin{align*}
    u_{p_i, out} &= u_{p_i, in} \\
    v_{p_i, out} &= v_{p_i, in} \\
    y_{p_i, out} &= y_{p_i, in} \text{, if } c_{p_i} = 0 \text{ or } p_i \in \text{in-range pixel, else }\nonumber
\end{align*}
\]

\[
\begin{align*}
    u_{p_i, out} &= u_{p_i, in} \times s f_{p_i} \\
    v_{p_i, out} &= v_{p_i, in} \times s f_{p_i} \\
    y_{p_i, out} &= l_{p_i, output} \text{. } (13)
\end{align*}
\]

**Scaling Factor – Advanced Mode**

The out-of-range pixel values can be mapped inwards according to how far they are from the boundary from the following equation:

\[ c_{p_i, output} = c_{R_{p_i}} + (c_{p_i} - c_{R_{p_i}}) \times \frac{d_{p_i, final}}{d(R_{p_i}, p_i)} \]  

(14)

\[ l_{p_i, output} = l_{\rho p_i} + c_{p_i, output} \times m_{comp} \]

Where \[ c_{R_{p_i}} \] is coming from the reference point as the origin of the linear transformation for compressing pixel \[ p_i \] as

Denote the reference point
$$R_{p_i} = (c_{R_{p_i}}, l_{R_{p_i}})$$

\[ l_{R_{p_i}} = \begin{cases} \max(l_{p_i}, l_v), & \text{if } l_{p_i} > l_v \\ \min(l_{p_i}, l_v), & \text{otherwise} \end{cases} \]  \hspace{1cm} (15)$$

\[ c_{R_{p_i}} = (l_{R_{p_i}} - l_{p_i}) \times \frac{1}{m_{comp}} \]
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.

**Restriction:** 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**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><strong>Header Present:</strong> Specifies whether the message includes a header phase. Must be zero for all Pixel Interpolator messages. Format = Enable</td>
<td></td>
</tr>
<tr>
<td>18:1</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><strong>SIMD Mode.</strong> Specifies the SIMD mode of the message being sent. Format = U1 0: SIMD8 mode 1: SIMD16 mode</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><strong>Interpolation Mode.</strong> Specifies which interpolation mode is used. Format = U1 0: Perspective Interpolation 1: Linear Interpolation <strong>Programming Note:</strong> This field cannot be set to &quot;Linear Interpolation&quot; unless <strong>Non-Perspective Barycentric Enable</strong> in 3DSTATE_CLIP is enabled.</td>
<td></td>
</tr>
<tr>
<td>13:2</td>
<td><strong>Message Type.</strong> Specifies the type of message being sent when pixel-rate evaluation requested. Format = U2 0: Per Message Offset (eval_snapped with immediate offset) 1: Sample Position Offset (eval_sindex) 2: Centroid Position Offset (eval_centroid) 3: Per Slot Offset (eval_snapped with register offset)</td>
<td></td>
</tr>
</tbody>
</table>

**Project:**
### Bits

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: When &quot;eval_centroid&quot; is selected and Render Target Independent Rasterization is enabled, HW may produce incorrect results.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:4</td>
<td><strong>Per Message Y Pixel Offset</strong></td>
</tr>
<tr>
<td></td>
<td>Specifies the Y Pixel Offset that applies to all slots.</td>
</tr>
<tr>
<td></td>
<td>Format = S4 2’s complement representing units of 1/16 pixel.</td>
</tr>
<tr>
<td></td>
<td>Range = [-8/16, +7/16]</td>
</tr>
</tbody>
</table>

| 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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:4</td>
<td><strong>Sample Index</strong></td>
</tr>
<tr>
<td></td>
<td>Specifies the sample index that applies to all slots.</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td></td>
<td>Range = [0,15]</td>
</tr>
<tr>
<td>3:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

**Message Payload for Most Messages**

This message payload applies to the following message types:

- Per Message Offset
- Sample Position Offset
- Centroid Position Offset

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7:0</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

**SIMD8 Per Slot Offset Message Payload**

This message payload applies only to the SIMD8 Per Slot Offset message type. The message length is 2.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td><strong>Slot 7 X Pixel Offset</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the X pixel offset for slot 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S4 2’s complement representing units of 1/16 pixel. The upper 28 bits are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [-8/16, +7/16]</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td><strong>Slot 6 X Pixel Offset</strong></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td><strong>Slot 5 X Pixel Offset</strong></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td><strong>Slot 4 X Pixel Offset</strong></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td><strong>Slot 3 X Pixel Offset</strong></td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td><strong>Slot 2 X Pixel Offset</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Slot 1 X Pixel Offset</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>Slot 0 X Pixel Offset</td>
</tr>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Slot 7 Y Pixel Offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the Y pixel offset for slot 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S4 2’s complement representing units of 1/16 pixel. The upper 28 bits are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [-8/16, +7/16]</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Slot 6 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Slot 5 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td>Slot 4 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Slot 3 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Slot 2 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Slot 1 Y Pixel Offset</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td>Slot 0 Y Pixel Offset</td>
</tr>
</tbody>
</table>

**SIMD16 Per Slot Offset Message Payload**

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td>Slot 7 X Pixel Offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the X pixel offset for slot 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S4 2’s complement representing units of 1/16 pixel. The upper 28 bits are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [-8/16, +7/16]</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>Slot 6 X Pixel Offset</td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td>Slot 5 X Pixel Offset</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Slot 4 X Pixel Offset</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Slot 3 X Pixel Offset</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td>Slot 2 X Pixel Offset</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Slot 1 X Pixel Offset</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>Slot 0 X Pixel Offset</td>
</tr>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Slot 15 X Pixel Offset</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Slot 14 X Pixel Offset</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Slot 13 X Pixel Offset</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td>Slot 12 X Pixel Offset</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Slot 11 X Pixel Offset</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Slot 10 X Pixel Offset</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Slot 9 X Pixel Offset</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td>Slot 8 X Pixel Offset</td>
</tr>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td>Slot 7 Y Pixel Offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the Y pixel offset for slot 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [-8/16, +7/16]</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td>Slot 6 Y Pixel Offset</td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td>Slot 5 Y Pixel Offset</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td>Slot 4 Y Pixel Offset</td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td>Slot 3 Y Pixel Offset</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td>Slot 2 Y Pixel Offset</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td>Slot 1 Y Pixel Offset</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td>Slot 0 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.7</td>
<td>31:0</td>
<td>Slot 15 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.6</td>
<td>31:0</td>
<td>Slot 14 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.5</td>
<td>31:0</td>
<td>Slot 13 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.4</td>
<td>31:0</td>
<td>Slot 12 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.3</td>
<td>31:0</td>
<td>Slot 11 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.2</td>
<td>31:0</td>
<td>Slot 10 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.1</td>
<td>31:0</td>
<td>Slot 9 Y Pixel Offset</td>
</tr>
<tr>
<td>M3.0</td>
<td>31:0</td>
<td>Slot 8 Y Pixel Offset</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 7 Format = IEEE_Float</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 6</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 5</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 4</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 3</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 2</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 1</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 0</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 7 Format = IEEE_Float</td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 6</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 5</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 4</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 3</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 2</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 1</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 0</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 6</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 5</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 4</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 3</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 2</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 1</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 0</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
</tr>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 6</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 5</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 4</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 3</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 2</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 1</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
</tr>
<tr>
<td>W2.7</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 15</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>W2.6</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 14</td>
</tr>
<tr>
<td>W2.5</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 13</td>
</tr>
<tr>
<td>W2.4</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 12</td>
</tr>
<tr>
<td>W2.3</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 11</td>
</tr>
<tr>
<td>W2.2</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 10</td>
</tr>
<tr>
<td>W2.1</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 9</td>
</tr>
<tr>
<td>W2.0</td>
<td>31:0</td>
<td>Barycentric[1] for Slot 8</td>
</tr>
<tr>
<td>W3.7</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 15</td>
</tr>
<tr>
<td>W3.6</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 14</td>
</tr>
<tr>
<td>W3.5</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 13</td>
</tr>
<tr>
<td>W3.4</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 12</td>
</tr>
<tr>
<td>W3.3</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 11</td>
</tr>
<tr>
<td>W3.2</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 10</td>
</tr>
<tr>
<td>W3.1</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 9</td>
</tr>
<tr>
<td>W3.0</td>
<td>31:0</td>
<td>Barycentric[2] for Slot 8</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Project</th>
<th>URB Size</th>
<th>URB Rows</th>
<th>URB Rows when SLM Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The CPU cannot read the URB directly.</td>
</tr>
</tbody>
</table>
**URB State**

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Header Required</th>
<th>Shared Local Memory Support</th>
<th>Stateless Support</th>
<th>Address Modes</th>
<th>Vector Width</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>URB Read HWORD</td>
<td>yes</td>
<td>N/A</td>
<td>N/A</td>
<td>handle + URBoffset or handle + URBoffset + offset</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>URB Write HWORD</td>
<td>yes</td>
<td>N/A</td>
<td>N/A</td>
<td>handle + URBoffset or handle + URBoffset + offset</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>URB Read OWORD</td>
<td>yes</td>
<td>N/A</td>
<td>N/A</td>
<td>handle + URBoffset or handle + URBoffset + offset</td>
<td>1, 2</td>
<td></td>
</tr>
</tbody>
</table>
"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**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><strong>Header Present.</strong> This bit must be 1 for all URB messages.</td>
</tr>
<tr>
<td>18:17</td>
<td>Ignored</td>
</tr>
<tr>
<td>16</td>
<td><strong>Per Slot Offset.</strong> 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. <strong>Programming Note:</strong> This bit must be 0 for URB_ATOMIC_* messages.</td>
</tr>
<tr>
<td>15</td>
<td><strong>Complete.</strong> For URB_WRITE*, URB_SIMD8_WRITE and URB_ATOMIC*, this bit is ignored. For URB_READ* and URB_SIMD8_READ, if set, this bit signals that the thread is finished reading from the URB entries referenced by the handles, causing the entries to be deallocated. This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose.</td>
</tr>
</tbody>
</table>
### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td><strong>Swizzle Control.</strong> This field is used to specify which &quot;swizzle&quot; operation is to be performed on the write data. It indirectly specifies whether one or two handles are valid.</td>
</tr>
<tr>
<td></td>
<td>0: URB_NOSWIZZLE. The message accesses a single URB entry (using <strong>URB Handle 0</strong>).</td>
</tr>
<tr>
<td></td>
<td>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 <strong>URB Handle 1</strong>, and the lower DWords (3:0) contain data associated with <strong>URB Handle 0</strong>.</td>
</tr>
</tbody>
</table>

| 13:3 | **Global Offset.** This field specifies a destination offset (in 256-bit units) from the start of the URB entries, as referenced by **URB Handle n**, at which the data (if any) is written or read. |
|      | When URB_INTERLEAVED is used, this field provides a 256-bit granular offset applied to both URB entries. |
|      | If the **Per Slot Offset** bit is set, this offset is added to the per-slot offsets in the header to obtain the overall offset. |
|      | For the URB_*_OWORD messages, this offset is in 128-bit units instead of 256-bit units. |
|      | For the URB_ATOMIC* messages, this offset is in 32-bit units instead of 256-bit units. |
|      | Format = U11 |
|      | Range = [0, 1023] for URB_*_HWORD messages. |
|      | Range = [0, 2047] for URB_*_OWORD messages. |
|      | Range = [0, 2047] for URB_ATOMIC* messages. |

| 2:0  | **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: Reserved |

### URB_WRITE and URB_READ

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

**Message Header**

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:17</td>
<td>Ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td><strong>High OWORD Enable</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For URB_READ_OWORD and</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>URB_WRITE_OWORD with NOSWIZZLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates whether the 128 bits of the GRF register is used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: 1 OWord, read into or written from the low 128 bits of the GRF register.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: 1 OWord, read into or written from the high 128 bits of the GRF register.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When <strong>Swizzle Control</strong> = URB_INTERLEAVED this bit controls Vertex 1 DATA[3].</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When <strong>Swizzle Control</strong> = URB_NOSWIZZLE this bit controls Vertex 0 DATA[7].</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is ANDed with the corresponding channel enable to determine the final channel enable. For the URB_READ_OWORD &amp; 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 &amp; URB_WRITE_HWORD messages, when final channel enable is 1 it indicates that Vertex 1 DATA [3] will be written to the surface.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>Vertex 1 DATA [2] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Vertex 1 DATA [1] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Vertex 1 DATA [0] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Vertex 0 DATA [3] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Vertex 0 DATA [2] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Vertex 0 DATA [1] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Vertex 0 DATA [0] Channel Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td><strong>Slot 1 Offset.</strong> This field, after adding to the Global Offset field in the message descriptor, specifies the offset (in 256-bit units) from the start of the URB entry, as referenced by URB Handle 1, at which the data will be accessed. This field is ignored unless Swizzle Control is set to URB_INTERLEAVED. For the URB_<em><em>OWORD messages, this offset is in 128-bit units instead of 256-bit units. Format = U32 Range = [0, 1023] for URB</em></em><em>HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined. Range = [0, 2047] for URB</em>*_OWORD messages. The range of the calculated offset must fall within the range [0, 2047] or behavior is undefined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td><strong>Slot 0 Offset.</strong> This field, after adding to the Global Offset field in the message descriptor, specifies the offset (in 256-bit units) from the start of the URB entry, as referenced by URB Handle 0, at which the data will be accessed. For the URB_<em><em>OWORD messages, this offset is in 128-bit units instead of 256-bit units. Format = U32 Range = [0, 1023] for URB</em></em>_HWORD messages. The range of the calculated offset must fall within the range [0, 1023] or behavior is undefined.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Payload</th>
<th>Usage</th>
</tr>
</thead>
</table>

### Table

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>or behavior is undefined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = ([0, 2047]) for URB_*_OWORD messages. The range of the calculated offset must fall within the range ([0, 2047]) or behavior is undefined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>31:16</td>
<td><strong>GS Number of Output Vertices for Slot 1.</strong> Indicates the number of vertices output for geometry shader slot 1 primitive. This field is only defined if end-of-thread is set on the message. It is ignored for all messages from non-GS threads. Format = U16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>GS Number of Output Vertices for Slot 0.</strong> Indicates the number of vertices output for geometry shader slot 0 primitive. This field is only defined if end-of-thread is set on the message. It is ignored for all messages from non-GS threads. Format = U16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.1</td>
<td>31:16</td>
<td><strong>Handle ID 1.</strong> 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 <strong>Swizzle Control</strong> indicates Interleave mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>URB Handle 1.</strong> This is the URB handle where channel 1’s results are to be written or read. This field is ignored unless <strong>Swizzle Control</strong> indicates interleave mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.0</td>
<td>31:16</td>
<td><strong>Handle ID 0.</strong> 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 <strong>Swizzle Control</strong> indicates Interleave mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>URB Handle 0.</strong> This is the URB handle where channel 0’s results are to be written or read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td>Usage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>URB_NOSWIZZLE</td>
<td>The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The <strong>Swizzle Control</strong> field of the message descriptor must be set to 'NoSwizzle'.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>URB_INTERLEAVED</td>
<td>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 <strong>Swizzle Control</strong> field of the message descriptor must be set to 'Interleave'.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**URB_NOSWIZZLE**

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

**Programming Notes:**
- The URB function *will use* (not ignore) the Channel Enables associated with this message.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td><strong>Vertex Data [7]</strong></td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td><strong>Vertex Data [6]</strong></td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td><strong>Vertex Data [5]</strong></td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td><strong>Vertex Data [4]</strong></td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td><strong>Vertex Data [3]</strong></td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td><strong>Vertex Data [2]</strong></td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td><strong>Vertex Data [1]</strong></td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td><strong>Vertex Data [0]</strong></td>
</tr>
<tr>
<td>M2.7</td>
<td>31:0</td>
<td><strong>Vertex Data [15]</strong></td>
</tr>
<tr>
<td>M2.6</td>
<td>31:0</td>
<td><strong>Vertex Data [14]</strong></td>
</tr>
<tr>
<td>M2.5</td>
<td>31:0</td>
<td><strong>Vertex Data [13]</strong></td>
</tr>
<tr>
<td>M2.4</td>
<td>31:0</td>
<td><strong>Vertex Data [12]</strong></td>
</tr>
<tr>
<td>M2.3</td>
<td>31:0</td>
<td><strong>Vertex Data [11]</strong></td>
</tr>
<tr>
<td>M2.2</td>
<td>31:0</td>
<td><strong>Vertex Data [10]</strong></td>
</tr>
<tr>
<td>M2.1</td>
<td>31:0</td>
<td><strong>Vertex Data [9]</strong></td>
</tr>
<tr>
<td>M2.0</td>
<td>31:0</td>
<td><strong>Vertex Data [8]</strong></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Mn.7</td>
<td>31:0</td>
<td><strong>Vertex Data [8(n-1)+7]</strong></td>
</tr>
<tr>
<td>Mn.6</td>
<td>31:0</td>
<td><strong>Vertex Data [8(n-1)+6]</strong></td>
</tr>
</tbody>
</table>
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 Restrictions:**
- The URB function *will use* (not ignore) the Channel Enables associated with this message.
- Writes to overlapping addresses of vertex0 and vertex1 will have undefined write ordering.
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.

<table>
<thead>
<tr>
<th>Payload</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>URB_NOSWIZZLE</td>
<td>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'.</td>
</tr>
<tr>
<td>URB_INTERLEAVED</td>
<td>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'.</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Vertex Data [7]</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Vertex Data [6]</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Vertex Data [5]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Vertex Data [4]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Vertex Data [3]</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Vertex Data [2]</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Vertex Data [1]</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Vertex Data [0]</td>
</tr>
<tr>
<td>W1.7</td>
<td>31:0</td>
<td>Vertex Data [15]</td>
</tr>
</tbody>
</table>
The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing $n$ 4-DWord vertex elements ($n > 1$).

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1.6</td>
<td>31:0</td>
<td>Vertex Data [14]</td>
</tr>
<tr>
<td>W1.5</td>
<td>31:0</td>
<td>Vertex Data [13]</td>
</tr>
<tr>
<td>W1.4</td>
<td>31:0</td>
<td>Vertex Data [12]</td>
</tr>
<tr>
<td>W1.3</td>
<td>31:0</td>
<td>Vertex Data [11]</td>
</tr>
<tr>
<td>W1.2</td>
<td>31:0</td>
<td>Vertex Data [10]</td>
</tr>
<tr>
<td>W1.1</td>
<td>31:0</td>
<td>Vertex Data [9]</td>
</tr>
<tr>
<td>W1.0</td>
<td>31:0</td>
<td>Vertex Data [8]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Wn.7</td>
<td>31:0</td>
<td>Vertex Data [8n+7]</td>
</tr>
<tr>
<td>Wn.6</td>
<td>31:0</td>
<td>Vertex Data [8n+6]</td>
</tr>
<tr>
<td>Wn.5</td>
<td>31:0</td>
<td>Vertex Data [8n+5]</td>
</tr>
<tr>
<td>Wn.4</td>
<td>31:0</td>
<td>Vertex Data [8n+4]</td>
</tr>
<tr>
<td>Wn.3</td>
<td>31:0</td>
<td>Vertex Data [8n+3]</td>
</tr>
<tr>
<td>Wn.2</td>
<td>31:0</td>
<td>Vertex Data [8n+2]</td>
</tr>
<tr>
<td>Wn.1</td>
<td>31:0</td>
<td>Vertex Data [8n+1]</td>
</tr>
<tr>
<td>Wn.0</td>
<td>31:0</td>
<td>Vertex Data [8n+0]</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Payload</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>URB_NOSWIZZLE</td>
<td>The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The <strong>Swizzle Control</strong> field of the message descriptor must be set to 'NoSwizzle'.</td>
</tr>
<tr>
<td>URB_INTERLEAVED</td>
<td>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 <strong>Swizzle Control</strong> field of the message descriptor must be set to 'Interleave'.</td>
</tr>
</tbody>
</table>

### URB_NOSWIZZLE

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

**Programming Notes:**

- The URB function will use (not ignore) the Channel Enables associated with this message.

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>------------------</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [3]</strong></td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [2]</strong></td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [1]</strong></td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [0]</strong></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [3]</strong></td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [2]</strong></td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [1]</strong></td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [0]</strong></td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

**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 Restrictions:**

- The URB function *will use* (not ignore) the Channel Enables associated with this message.
- Writes to overlapping addresses of vertex0 and vertex1 will have undefined write ordering.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.7</td>
<td>31:0</td>
<td><strong>Vertex 1 Data [3]</strong></td>
</tr>
<tr>
<td>M1.6</td>
<td>31:0</td>
<td><strong>Vertex 1 Data [2]</strong></td>
</tr>
<tr>
<td>M1.5</td>
<td>31:0</td>
<td><strong>Vertex 1 Data [1]</strong></td>
</tr>
<tr>
<td>M1.4</td>
<td>31:0</td>
<td><strong>Vertex 1 Data [0]</strong></td>
</tr>
<tr>
<td>M1.3</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [3]</strong></td>
</tr>
<tr>
<td>M1.2</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [2]</strong></td>
</tr>
<tr>
<td>M1.1</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [1]</strong></td>
</tr>
<tr>
<td>M1.0</td>
<td>31:0</td>
<td><strong>Vertex 0 Data [0]</strong></td>
</tr>
</tbody>
</table>
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 Restrictions:

- **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.

<table>
<thead>
<tr>
<th>Payload</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>URB_NOSWIZZLE</td>
<td>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'.</td>
</tr>
<tr>
<td>URB_INTERLEAVED</td>
<td>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'.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td><strong>Vertex Data [3]</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td><strong>Vertex Data [2]</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td><strong>Vertex Data [1]</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td><strong>Vertex Data [0]</strong></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td><strong>Vertex Data [3]</strong></td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td><strong>Vertex Data [2]</strong></td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td><strong>Vertex Data [1]</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td><strong>Vertex Data [0]</strong></td>
</tr>
</tbody>
</table>
URB_INTERLEAVED

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7</td>
<td>31:0</td>
<td>Vertex Data [3]</td>
</tr>
<tr>
<td>W0.6</td>
<td>31:0</td>
<td>Vertex Data [2]</td>
</tr>
<tr>
<td>W0.5</td>
<td>31:0</td>
<td>Vertex Data [1]</td>
</tr>
<tr>
<td>W0.4</td>
<td>31:0</td>
<td>Vertex Data [0]</td>
</tr>
<tr>
<td>W0.3</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.2</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.1</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Reserved (not written to GRF)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>URB Opcode</th>
<th>new_dst</th>
<th>ret</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>URB_ATOMIC_MOV</td>
<td>0</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>URB_ATOMIC_INC</td>
<td>old_dst + 1</td>
<td>old_dst</td>
<td></td>
</tr>
<tr>
<td>URB_ATOMIC_ADD</td>
<td>old_dst + src0</td>
<td>old_dst</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td>Source0 Data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the source 0 data for the atomic operation. This field is ignored for the URB_ATOMIC_INC message. Format = U32</td>
<td></td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>M0.0</td>
<td>31:16</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15:0 URB Handle. The URB handle to access.</td>
<td></td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:1</td>
<td></td>
<td>Reserved (not written to GRF)</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:0</td>
<td>Return Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the value of the return data for the atomic operation. Format = U32</td>
</tr>
</tbody>
</table>
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.

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 N0 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.

### Project:
- **MMIO Read/Write**: allows a message to read or write an MMIO register. The MEDIA_VFE_STATE command has a field which limits the accesses for security.

**Message Descriptor**

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><strong>Header Present.</strong> This bit must be 0 for all Message Gateway messages.</td>
</tr>
<tr>
<td>18:17</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>
| 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. |
<p>| 14   | <strong>AckReq.</strong> 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 |</p>
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-11</td>
<td>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 null 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.</td>
</tr>
<tr>
<td>13:3</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>2:0</td>
<td><strong>SubFuncID.</strong> Identify the supported sub-functions by Message Gateway. Encodings are: 000b: <strong>OpenGateway.</strong> Open the gateway for the requester thread. 001b: <strong>CloseGateway.</strong> Close the gateway for the requester thread. 010b: <strong>ForwardMsg.</strong> Forward the formatted message to the recipient thread with the given offset from the recipient’s register base. 011b: <strong>GetTimeStamp.</strong> Read absolute and relative timestamps. 100b: <strong>BarrierMsg.</strong> Record an additional thread reaching the barrier. 101b: <strong>UpdateGatewayState.</strong> Update the barrier byte for a barrier. 110b: <strong>MMIO Read/Write.</strong> 111b: Reserved.</td>
</tr>
</tbody>
</table>

**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.

**Project:**

The BarrierMsg command does not use an OpenGateway message.
## Message Payload

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.5</td>
<td>31:29</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28:21</td>
<td><strong>RegBase</strong>: 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]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20:11</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10:8</td>
<td><strong>Gateway Size</strong>: 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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Dispatch ID</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:16</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Reserved: MBZ.</td>
<td></td>
</tr>
<tr>
<td>M0.3:0</td>
<td>31:0</td>
<td>Ignored</td>
<td></td>
</tr>
</tbody>
</table>

## Writeback Message to Requester Thread

The writeback message is only sent if the `AckReq` bit in the message descriptor is set.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:1</td>
<td>31:0</td>
<td>Reserved (not overwritten)</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:20</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Shared Function ID.</strong> The message gateway’s shared function ID.</td>
</tr>
<tr>
<td></td>
<td>15:3</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>2:0</td>
<td><strong>Error Code</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000b: <strong>Successful.</strong> No Error (Normal).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101b: <strong>Opcode Error.</strong> Attempt to send a message which is not either open/close/forward.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other codes: Reserved.</td>
</tr>
</tbody>
</table>

**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.

**Project:**

The BarrierMsg command does not use a CloseGateway message.

**Message Payload**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7:6</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.5</td>
<td>31:8</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Dispatch ID:</strong> 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.</td>
</tr>
<tr>
<td>M0.4:0</td>
<td></td>
<td>Ignored</td>
</tr>
</tbody>
</table>

**Writeback Message to Requester Thread**

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.
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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:29</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>28:16</td>
<td><strong>Offset</strong>: 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.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Offset[28:21]: Register offset from the gateway base (Range [0, 127]: bit 12 MBZ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Offset[20:18]: DW offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Offset[17:16]: Byte offset (must be 00 for all DW length cases)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Programming restriction</strong>: R0 can not be used as destination GRF register for ForwardMsg. NULL register is also not allowed as destination.</td>
</tr>
<tr>
<td>15:11</td>
<td></td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>10:8</td>
<td></td>
<td><strong>Length</strong>: The length of the data segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000: 1 byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001: 1 word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010: 1 dword</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011: 2 dwords</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100: 4 dwords</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101-111: Reserved</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>Dispatch ID</strong>: 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is ignored by Message Gateway</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>M0.4</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>31:30</td>
<td></td>
<td><strong>SliceID</strong>: The Slice ID as part of the Recipient field is used to identify the slice containing the EU to whom the message is forwarded.</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td><strong>HalfSliceID</strong>: The Half-slice ID is used to identify the half-slice containing the EU to whom the message is forwarded. For project before HSW the half-slice ID is encoded in the EUID.</td>
</tr>
<tr>
<td>31:30</td>
<td></td>
<td><strong>EUID</strong>: The Execution Unit ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.</td>
</tr>
<tr>
<td>29:28</td>
<td></td>
<td><strong>TID</strong>: The Thread ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.</td>
</tr>
</tbody>
</table>
### DWord 15:0

**Key**
The key to match with the one stored in the recipient thread's entry in Message Gateway. 

[DevSNB+] Ignored

### DWord 31:0 - M0.2

**Data Segment DWord 2:** valid only for the 4-DWord data segment length

### DWord 31:0 - M0.1

**Data Segment Dword 1:** valid only for the 2- and 4-DWord data segment lengths

### DWord 31:0 - M0.0

**Data Segment DWord 0:** valid only for the 1-, 2- and 4-Dword data segment lengths

**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.

### DWord 23:16

**Data Segment Byte 0**

### DWord 15:8

**Data Segment Byte 0**

### DWord 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:1</td>
<td>31:0</td>
<td>Reserved (not overwritten)</td>
</tr>
<tr>
<td>W0.0</td>
<td>31:20</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Shared Function ID.</strong> The message gateway’s shared function ID.</td>
</tr>
<tr>
<td></td>
<td>15:3</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>2:0</td>
<td><strong>Error Code</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000b: <strong>Successful.</strong> No Error (Normal).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001b: Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010b: <strong>Gateway Closed.</strong> Attempt to send a message through a closed gateway.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011b: Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100b: Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101b: <strong>Opcode Error.</strong> Attempt to send a message which is not either open/close/forward.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110b: <strong>Invalid Message Size.</strong> Attempt to forward a message with length greater than 4 DWords.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td>Return to High GRF:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: the return 128-bit data goes to the first half of the destination GRF register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: the return 128-bit data goes to the second half of the destination GRF register</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.5</td>
<td>31</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>30:8</td>
<td>Reserved : MBZ</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is ignored by Message Gateway</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:4</td>
<td></td>
<td>Reserved (not overwritten)</td>
</tr>
<tr>
<td>W0.3 31:0</td>
<td><strong>RelativeTimeLapHigh</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>W0.2 31:20</td>
<td><strong>RelativeTimeLapLow</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>Reserved : MBZ</td>
<td></td>
</tr>
<tr>
<td>W0.1 31:0</td>
<td><strong>AbsoluteTimeLapHigh</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>W0.0 31:20</td>
<td><strong>AbsoluteTimeLapLow</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>Reserved : MBZ</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7 31:0</td>
<td><strong>RelativeTimeLapHigh</strong></td>
<td></td>
</tr>
<tr>
<td>W0.6 31:20</td>
<td><strong>RelativeTimeLapLow</strong></td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>Reserved : MBZ</td>
<td></td>
</tr>
<tr>
<td>W0.5 31:0</td>
<td><strong>AbsoluteTimeLapHigh</strong></td>
<td></td>
</tr>
</tbody>
</table>
**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 mark the Barrier requester thread. There is no immediate response from the Message Gateway. When the counter value equates **Barrier Thread Count**, Message Gateway will send response back to all the Barrier requesters.

**Message Payload**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td>M0.2</td>
<td>31</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td></td>
<td>30:28</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td><strong>BarrierID</strong>: This field indicates which one from the 16 Barrier States is updated. Format: U4 Note: this field location matches with that of R0 header.</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>Ignored</strong></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td><strong>Barrier Count Enable</strong>: Allows the message to reprogram the barrier count. If set, the current value of the barrier state is compared to the Barrier Count field (below). If these values are equal, the barrier is considered satisfied, barrier responses are sent to the waiting thread(s) including the sending thread, and the barrier state is reset to 0. If these values are not equal, the</td>
</tr>
</tbody>
</table>
barrier state is incremented and the sending thread is added to the list of threads waiting on this barrier.

If clear, the Message Gateway increments the Barrier counter and marks the Barrier requester thread. There is no immediate response from the Gateway. When the counter value equates Barrier Thread Count, Gateway will send response back to all the Barrier requesters.

Format: Enable

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
|       | 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 only sent if the **AckReq** bit in the message descriptor is set.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:1</td>
<td></td>
<td><strong>Reserved (not overwritten)</strong></td>
</tr>
<tr>
<td>W0.0</td>
<td>31:20</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Shared Function ID.</strong> Contains the message gateway's shared function ID.</td>
</tr>
<tr>
<td></td>
<td>15:3</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td></td>
<td>2:0</td>
<td><strong>Error Code</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000: <strong>Successful. No Error (Normal).</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>001: <strong>Error (Barrier is inactive).</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other encodings are reserved.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.7:1</td>
<td>31:0</td>
<td><strong>Reserved (not overwritten)</strong></td>
</tr>
</tbody>
</table>
### MMIOReadWrite Message

MMIO read/write is not allowed to registers that are associated with a particular slice.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0.0</td>
<td>31:16</td>
<td>Reserved (not overwritten)</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Reserved (not overwritten)</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Reserved (not overwritten)</td>
</tr>
</tbody>
</table>

### Message Payload

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td></td>
</tr>
<tr>
<td>M0.5</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.3</td>
<td>31:1</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>MMIO R/W:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – MMIO Read – a response will be sent to the EU with read data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – MMIO Write – no response is sent to EU (unless acknowledge requested in sideband)</td>
</tr>
<tr>
<td>M0.2</td>
<td>31:28</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>22:0</td>
<td>MMIO Address:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The MMIO Byte address to be accessed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The bottom 2 bits must be zero.</td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:0</td>
<td>MMIO Write Data (Only if MMIO R/W = 1, otherwise ignored).</td>
</tr>
</tbody>
</table>

### Writeback Message to Requester Thread (MMIO Read Only)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.6</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.5</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.4</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.3</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.2</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------------</td>
</tr>
<tr>
<td>R0.1</td>
<td>31:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>R0.0</td>
<td>31:0</td>
<td><strong>MMIO Read Data</strong></td>
</tr>
</tbody>
</table>
Shared Functions - Media Sampler

This section describes the functionality of the Media Sampler.
**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 $\text{MaxNumMVs} = \text{MaxNumMVsMinusOne} + 1$ is less than 4, we will set minor MV flag off: $\text{MinorMVsFlag} = 0$, i.e. no minor motion vectors will be generated.

The reason of having this parameter $\text{MaxNumMVs}$ 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, $\text{MaxNumMVs}$ should be set to 32.

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

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

3) Final shape decision is done after BME.

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

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

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

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

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

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

The following partitions are not supported as alternative partition.

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

**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 \textbf{MaxNumMVs} (see pseudo-code below for detail).

Consequently, we have the best combined 8x8 solutions with \textbf{N} motion vectors for some \textbf{N} less or equal to \textbf{MaxNumMVs}.

Assume $\text{distA}[k][s]$ 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 $\text{distA}[k][s]$ is the bidirectional one of the corresponding bus-block and sub-shape. And assume some large number, say 128x16=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 \textbf{MaxNumMVs} is disabled, i.e. \textbf{MaxNumMVs}>16:

```plaintext
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][3]; s[k] = 2; }
        if ( distA[k][3]<d[k] ) { d[k] = distA[k][3]; s[k] = 3; }
    }
    *MinorShape = s[0] | (s[1]<<2) | (s[2]<<4) | (s[3]<<6);
}
```

Now for the case of using \textbf{MaxNumMVs} control:

```plaintext
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[10] = (d[0] + d[10] < d[8] + d[2]) ? 0x30 : 0x03;  // 5 mvs
    s[4] = 0;  // 2 mvs
    d[4] = d[0] + d[2];
    d[14] = 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 DistoMajor[k] <= BestDisto + IntTol.

**Shape Update after FME**

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

**Final Code Decision after BME**

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

Then, we choose the best among the improved candidate shapes.
**Integer Motion Estimation**

IME, the integer motion estimation, is the most key part of VME. In our current design, the minimal functional block is to do a full search over a search unit. This functional block is then called via two distinctive methods:

1. Via a predefined searching path of search units.
2. Via a dynamic process based on the previous results.

This section will describe both.

**Reference Window and Search Units**

The reference window is a rectangular region fetched put in the reference cache for VME. Either one or two reference windows are allowed to be loaded into the reference cache. In the case of dual windows, both windows follow a common search path or different paths (relative to their corresponding Start Center) depending on the dual search path option flag.

The total reference cache is limited to 2K bytes and only the luma component searching is performed. For example, we may select the reference windows to be one of the following sample choices: one 64x32 area, one 48x40 area, two 40x24 areas, or two 32x32 areas, where the possible reference address will cover an area of 48x16, 32x24, 24x8, and 16x16.

As a convention, we will call the valid reference addressing region the *reference region*, and its width and height are called the *reference window width* and *height* respectively. So the reference loading region of VME has therefore 16 more columns and 16 more rows.

It is not efficient for hardware to search one location at a time due to reference cache access bandwidth and latency constraints. Thus, possible reference search locations are grouped in a predefined pattern, and all locations within the same group must be either all are chosen or all are skipped. These predefined groups are called *search unit* (SU). *Reference Window and Search Units* shows a sample of grouping search locations into searching units. The reference window in the figure has a dimension of 32x20, and assuming the source block is 8x8, the dark dots indicate all legitimate reference locations for motion searching. It shows a partitioning of SUs of 16 locations.

In general, the indices of SUs are given by counting rows and units within the row.
Example of Search Units in a reference window

Given a fixed reference cache access latency, SU size is determined solely based on the source block size as shown in *Reference Window and Search Units*. Note that SU sizes for both 16x8 and 8x16 source blocks are both 8x4, which gives a preference for motions along horizontal direction.

<table>
<thead>
<tr>
<th>Source Block Dimension</th>
<th>Search Unit (SU) Size (X x Y)</th>
<th>GT Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x16</td>
<td>4x4</td>
<td>Y</td>
</tr>
<tr>
<td>16x8</td>
<td>8x4</td>
<td>Y</td>
</tr>
<tr>
<td>8x16</td>
<td>8x4</td>
<td>N</td>
</tr>
<tr>
<td>8x8</td>
<td>8x8</td>
<td>Y</td>
</tr>
</tbody>
</table>

To keep tracking on whether a SU has been searched or not, an equivalent hardware process is implemented performing as a *search record* that marks whether any search units being searched is a bit- plan of the bit-length equal to the maximal index of SUs. Before searching starts, the search record must be reset which sets the value 0 (= yet-to-be-searched) to all legitimate SU indices, and the value 1 (= no longer available) for other SUs that are not intended to be searched.

Given a search window, unique indices are assigned to all SUs. A search path (SP), is a sequence of such indices. The number of SUs in a SP is called the length of the walker (denoted by LenSP here), which shall be a number more than one. Instead of storing the absolute indices of a search path, relative *search unit deltas* are sent instead. In the current VME a search unit delta is a 8 bit index consisting of a pair of 4-bit signed integers in [-8,8).
Given start center in a pair of 4-bit unsigned integer \((sx, sy)\), and denote a search path described in SU deltas \((dx[i],dy[i])\). The first search unit SU[0] will be the search unit which has the first reference address \((sx*4, sy*4)\) in integer-pel relative to the reference origin., i.e.

\[
SU[0].x = sx*4, \text{ and } SU[0].y = sy*4.
\]

The second search unit SU[1] is derived by adding \((dx[0],dy[0])\):

\[
SU[1].x = SU[0].x + dx[0]*4, \text{ and } SU[1].y = SU[0].y + dy[0]*4.
\]

In general, we have:

\[
SU[i+1].x = SU[i].x + dx[i]*4, \text{ and } SU[i+1].y = SU[i].y + dy[i]*4.
\]

When SU[i] is out of range, it is either always skipped or always wrapped depending on the SU wrapping flag.

When the SU wrapping flag is on, it is equivalent to as we perform

\[
SU[i].x = SU[i].x (\text{mod ref_win_width}), \text{ and } SU[i].y = SU[i].y (\text{mod ref_win_height}),
\]

As a convention, a NULL delta marks the end of the search path.

**Fixed and Adaptive Search Paths**

A fixed pattern motion search algorithm is an algorithm following some predefined SP with the designated MaxNumSU (maximal number of search units) less than or equal to LenSP (the fixed search path length). This is referred to as fixed pattern searching or predetermined searching.

When MaxNumSU > LenSP, (the maximal number of SU is more than what are given by the SP), the searching continues unless reaching a local minimum, which is called dynamic searching or adaptive searching or gradient searching. In this case, the current best result is used. If it is located in some SU boundary, the neighborhood SUs are checked and any one of them that is yet-to-be-searched will be the next SU to be searched. If all neighbor SUs are done, the process of IME is done. Fixed and Adaptive Search Paths illustrates on how neighbor SUs are defined for dynamic search.

Hardware maintains one scoreboard per reference to keep track of the state of SUs, whether being searched or yet-to-be-searched. When dual records are enabled on a single reference, both records share the same scoreboard for the reference.

**Sample Neighborhood SUs in a Dynamic Search**
*Fixed and Adaptive Search Paths* shows the algorithm of this integrated solution. In order to hide the decision logic of dynamic walking, the one-step-delayed-queue is implemented. So when searching the current SU, the next SU is put in the queue. If there are more SUs yet to be searched in the current SP, the next SU is the next SU according to SP; if there is no more SU from SP, the first unsearched neighbor SU (in some predefined order) based on the current best result will be put instead, and if there is no more unsearched neighbor SU, the integer searching terminates.

To reduce the one-step-delay, and to support bidirectional, we create the dual mode that allows the above algorithm to be ping-ponged between two search paths. *Fixed and Adaptive Search Paths* illustrates this case.

In both figures, the best MVs refer to the best resulting motion vectors so far. There are potentially total 41 motion vectors (1 for 16x16, 2 for 16x8, 2 for 8x16, 4 for 8x8, and 32 more for 8x4, 4x8, and 4x4 cases). **The current hardware implementation only considers the four 8x8 MVs.**

**VME in Single SP Mode**
VME in Dual SP Mode

Initialize VME

NextSU = WP[0],
K = 1;

ThisSU = NextSU;
Record(ThisSU) = 1;

Search Integer Motion Vectors in This SU

Yes

Compute

K < LenSP

Yes

Is the best MV
next to SU edge?

Yes

NextSU = SP[K];

No

Is there a neighbor
yet-to-be-searched?

Yes

NextSU = the neighbor SU;

No

NextSU = NULL;

End of Integer VME

No

Yes

K++;
K < MaxNumSU

NextSU = NULL;
**Fractional Motion Estimation**

Instead of following the exact interpolation as specified by the individual video standards, 4 tap interpolation is used for the Fractional Motion Estimation (FME) step in the VME engine. It is expected to be adjusted according to different standards.

**Interpolations**

Instead of following the exact interpolation as specified by the individual video standards, fixed 4 tap interpolation is used in the VME engine, as defined below:

1. \((-1, 5, 5, -1)/8\) for \(\frac{1}{2}\)-pel, i.e. \(s = (-P1 + P2*5 + P3*5 - P4 + 4)/8\) and
2. \((-1, 13, 5, -1)/16\) for \(\frac{1}{4}\)-pel position, i.e. \(c = (-P1 + P2*13 + P3*5 - P4 + 8)/16\).

**Fractional pixel locations**

The quarter-pels are actually the averages of its nearest integer and half pixel values. It is not hard to see our suggested interpolation formulas are very much the good approximations of the formulas from various standards.

For AVC, they should be the following 6-tap formulas in theory:

1. \((1, -5, 20, 20, -5, 1)/32\) for \(\frac{1}{2}\)-pel, i.e. \(s = ((P2 + P3)*20 - (P1 + P4)*5 + (F0 + F6))\), and
2. \((1, -5, 52, 20, -5, -1)/64\) for \(\frac{1}{4}\)-pel position.

For VC-1, the 4-tap filters are precisely defined:

1. \((-1, 9, 9, -1)/16\) for \(\frac{1}{2}\)-pel, and
2. \((-4, 53, 18, -3)/64\) for \(\frac{1}{4}\)-pel position.

In general, bilinear interpolation is accepted too:

1. \((0, 1, 1, 0)/2\) for \(\frac{1}{2}\)-pel (as used in MPEG2), and
2. \((0, 3, 1, 0)/4\) for \(\frac{1}{4}\)-pel position.

After IME is done, if the best Inter result is too bad, we may decide to stop the Inter-search to not waste the effort further computationally. If we decided to continue, we have the option to decide shape first to cut down FME calculation or to perform FME for all possible configurations.
VME performs the sub-block level intermediate shape decision first (see Shape Decision section for detail), then perform FME only for the reduced candidate shapes. In this way, the computation is reduced significantly with tunable small quality hit.

**8+8 vs. 7x7**

With a given sub-block of motion vector search, we also have multiple options to pursue the searching. Name two common extremes: 7x7 and 8+8.

Given an integer motion vector location, surrounding it there are 48 surrounding quarter-pel locations, and among them there are 8 are in half-pel grids. So we may check all 48, which covers the 7x7 region, for the best, or we may adopt a two-step approach by considering the half-pel grids first then followed by the second step of the quarter-pel refinement.

The one step method is named as 7x7, and the two step method called 8+8 as only 16 block comparisons are performed as shown in the next figure.

VME hardware follows the 8+8 approach.

**7x7 vs. 8+8, whereas the 8+8 method is used by VME**

**Partitioning Refinement**

When the partitioning refinement is enabled, the FME refinement results will be propagated to or sub-blocks as well, and a shape partitioning will be redone after the completion of both half-pel and quarter-pel searching for a possible better choice.

In the case when alternative candidate is enabled, both half-pel refinements are done in parallel, and then records are combined. Then, both quarter-pel refinements are done again in parallel, and combined
again prior to the final repartitioning. In HW implementation, we do the coarser on first, and the finer one later to achieve the above equivalence.

**BME and Weighted Prediction**

Bidirectional searching is performed to all candidate shapes.

A weighted bidirectional search is supported particularly for AVC implicit weighted prediction. Only a common subset of frame relations, which falls into linear interpolation with positive weight, is implemented. The weight between forward and backward is approximated into 5 cases only: 16 (quarter distance like Rf B X X Rb), 21 (one third distance like Rf B X Rb), 32 (half distance like Rf B Rb), 43 (two third distance like Rf X B Rb), and 48 (three quarter distance like RXXBR). Here the notation is for bidirectional prediction with display picture order, whereas Rf stands for forward reference, Rb for backward reference, B for the current bidirectional predicted picture and X is another picture in the sequence.

So if the forward prediction is \( \text{Ref0}[i] \), and the corresponding backward reference is \( \text{Ref1}[i] \), then the combined bidirectional motion prediction is calculated as the following:

\[
\text{Ref}[i] = ((64-\alpha) \times \text{Ref0}[i] + \alpha \times \text{Ref1}[i] + 32) >> 6;
\]

where, \( \alpha \) is one of the 5 weighting numbers mentioned above.

**Skip Check**

There are two SKIP modes:

- **SKIP_1MVP** – one MV pair for 16x16 macroblock, and
- **SKIP_4MVP** – four MV pairs for four 8x8 subblocks.

Otherwise, when Skip Check is enabled and the skip MV number does not exceed **MaxMumMV**, VME will first perform the fractional motion estimation at the skip centers provided by the motion vector pairs as specified by the corresponding mode. In this case the following distortions will be calculated:

1. **RawSkipDist** – (intended for AVC PB_Skip) the raw SAD/HAAR distortion calculated from the skip motion vectors with no costing added.
2. **NonSkipDist** – (intended for AVC B_Direct16x16) the adjusted non-skip distortion is defined by adding optionally the zero motion vector cost and 16x16 Inter mode penalty to **RawSkipDist**. And
3. **NonSkip8x8Dist[4]** – (intended for AVC B_Direct8x8) the four adjusted non-skip distortions for four individual 8x8 subblocks with the ZMV cost and 8x8q Inter mode penalty optionally added. (Note: This case may produce partitions with 8x8 subblock even if the 8x8 subblock shape is disabled.)

Optional implies whether add or not is purely depende on two enabling input bits: **NonSkipModeAdded** and **NonSkipMvAdded**. It should be also noted that **MODE_INTER_BWD** is not added to **NonSkipDist** or **NonSkip8x8Dist[4]** even though a skip center contains backward motion vector (this is for a direct
mode, whether the motion vector for a block is forward, backward or bidirection is derived from its spatial or temporal predictor and there is no coding cost).

If \texttt{RawSkipDist} is less than or equal to \texttt{EarlySkipSuccess} threshold, \texttt{MinDist} will be set to \texttt{RawSkipDist} if the skip MV number does not exceed \texttt{MaxMumMV}.

- If \texttt{EarlySuccessEn} flag is on, VME exits immediately after setting \texttt{MbSkipFlag} on, and \texttt{Direct8x8Pattern} = Fh.
- If \texttt{EarlySuccessEn} flag is off, VME continue the IME, FME, BME, and Direct8x8 searching after setting \texttt{MbSkipFlag} on and \texttt{Direct8x8Pattern} = Fh. VME will choose the skip output unless another better choice of code with less adjusted distortion is found.

If \texttt{RawSkipDist} is greater than \texttt{EarlySkipThreshold}, \texttt{MinDist} will be set to \texttt{NonSkipDist} if the skip MV number does not exceed \texttt{MaxMumMV}. \texttt{MbSkipFlag} will be always set to off. VME continue the IME, FME, BME, and Direct8x8 searchings. VME will still choose the skip output (with \texttt{MbSkipFlag} off) unless another better choice of code with less adjusted distortion is found.
**Direct 8x8 Search**

**Direct 8x8 Searching** and then possible replacement is performed ONLY for 16x16 source block.

**Direct 8x8 Searching** is performed only for **Skip_4MVP** mode when skip check is on to candidates of MbType in a partition that is in the Inter shape of 8x8 or minors, after IME, FME, and BME searchings.

For each candidate in 8x8 or smaller partition, and for each 8x8 sub-block, the corresponding codes will be replaced by the skip motion vector (pair) of the same 8x8 subblock, if all of the following requirements are satisfied:
• The non-skip 8x8 distortion \textbf{NonSkip8x8Dist[k]} is less than or equal to the adjusted 8x8 distortion of the corresponding codes.
• The merge does not violate uni-mix and bi-mix rules (the violating cases are skipped).
• The number of MVs used for the candidate adding the number of subblocks of the shape 8x8 must be less than or equal to \textbf{MaxNumMV}. Or otherwise it does not replace a uni-directional 8x8 MV with a true bi-directional skip MV pair.

Note that, during all of the above comparisons, we skip the process whenever the MV numbers exceeding the \textbf{MaxNumMV}.

If either \textbf{UniMixDisable} or \textbf{BiMixDis} is set, then there would be no direct8x8 block level replacement.

\textbf{Skip Check Only Mode}

\textbf{VME} supports the skip check only mode, when \textbf{Intra} is set off, \textbf{Inter} and \textbf{Skip} are enabled, and all 7 inter shapes (\textbf{SubMbPartMask}): 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, and 4x4 are set to 1 (all sub partitions are turned off). That indicates that none of the partition and sub partitions are valid for IME. Therefore IME is not performed, and no subsequent FME/BME is performed. This is another performance optimization choice if the intended usage is to check the skip centers only.

\textbf{Intra Prediction Estimation}

\textbf{Intra Prediction Estimation} state supports all Intra16x16, Intra8x8, and Intra4x4 modes. All predictions are based on original frame pixels for quick performance, as widely adopted in HW industry. There is a known quality drop.

For supporting AVS as well as providing finer knobs for AVC, five enabling flags are defined:

• Enable Intra16x16: whether Intra16x16 shall be performed.
• Enable Intra8x8: Enable all Intra8x8 modes, and the next flag determines which ones are actually performed.
• AVS Intra8x8 Flag: whether should perform the subset of 5 AVS modes or perform the super set of 8 AVC modes.
• Enable Intra4x4: Enable all Intra4x4 modes, and the next flag determines which ones are actually performed.
• AVS Intra4x4 Flag: whether should perform the subset of 5 AVS modes or perform the super set of 8 AVC modes.

\textbf{Transform Adjusted SAD}

A simple Wavelet transform, Haar transform, is used to refine the cost function measure of SAD. The per pixel difference goes through a 4x4 Haar transform. Then the SAD is replaced by the sum of the absolute values the transform domain coefficients (L1 norm) in the cost function. Haar transform here is used as a coarse estimation of the integer transform.

Assume the 4x4 block \textbf{Blk} is given in the following order:
The 4x4 Haar transform is performed using cascaded 2x2 Haar filters of the following steps:

- Four 4-tap row filter
- Two 4-tap column filter
- Two 2-tap row filter
- Two 2-tap column filter

Where the 2x2 Haar transform is given as:

\[
\begin{pmatrix}
1 & 1 \\
1 & -1
\end{pmatrix}
\]

This is equivalent to the following pseudo code functions:

```c
void Haar(short Blk4x4[16], short Haar4x4[16]) {
    short Tmp[16];
    // First level 4-element horizontal Haar for 4 rows:
    for ( int i=0; i<8; i++ ) {
        Haar4x4[8+i] = (Blk4x4[i*2] - Blk4x4[i*2+1]);
        // Storing LP 2x4 in scan order:
        Tmp[i] = (Blk4x4[i*2] + Blk4x4[i*2+1]);
    }
    // First level 4-element vertical Haar for 2 columns:
```
Haar4x4[4] = (Tmp[0] - Tmp[2]);

// Storing LP 2x2 in scan order:
Tmp[0] = (Tmp[0] + Tmp[2]);

// Second level 2-element horizontal Haar for 2 columns:
Haar4x4[3] = (Tmp[0] - Tmp[1]);

// Storing LP 1x2:
Tmp[0] = (Tmp[0] + Tmp[2]);

// Second level 2-element vertical Haar:
Haar4x4[1] = (Tmp[0] - Tmp[1]);
Haar4x4[0] = (Tmp[0] + Tmp[1]);

int AdjustedSAD(BYTE Src[16], BYTE Ref[16])
{
    short diff[16], diffH[16];
    for ( int i=0; i<16; i++ ) diff[i] = r[i]-s[i];
    Haar(diff, diffH);
    intasad = 0;
    for ( int i=0; i<16; i++ ) asad += (diff[i] < 0 ? -diff[i] : diff[i]);
    asad >>= (12 - DISTBIT4X4);
    return (asad);
}

Thus instead of calculating the SAD of the actual pixel values, now we apply SAD to the after transformation values.

As the Haar transform basis vectors have a magnitude of ½, instead of the normalized Haar of 1/sqrt(2), the resulting transformed coefficients maintain the same bit precision as the input. Thus the sum tree has the same precision as without the transform adjustment. However, this version of the Haar transform has low weightings on the DC and low AC terms, which may not be optimal as a motion-search cost function.

**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:

a) ESS: EarlySkipSuccess = Early successful return after Skip is checked
b) EIS: EarlyImeStop = Early IME stop when a good match is found inside of IME process.
b) ITG: ImeTooGood = Early successful return after IME is done when a good enough match is found.
a) ITB: ImeTooBad = Early termination do skip fractional and bidirectional refinement after IME is done with a hopelessly bad match as the best result.
c) EFS: EarlyFmeSuccess = Early Success after Fractional ME to skip bidirectional search.

Note. 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 0x3FF.
Performance Information

VME makes many internal decisions such as whether or not early exits occurred. Additionally, the number of search units processed and the total clocks spent per message are valuable to software for real-time adjustments or testing and statistical analysis. VME output message contains such information to fulfill this basic feature.

The output message for VME contains fields to encode decision and performance counters. This includes performed sub-functions (IME, FME, BME, etc), the early exit conditions, and other internal decisions.

Of the other internal decisions, there are fields for whether or not FME or BME improved the primary candidate. These bits will be set when FME or BME modifies the best mv decision. If the alternate partition or extra candidate results in a lower cost at the end of VME, a bit will be used to represent that the alternate beat the original best. Lastly, 1 bit will be used to indicate partitioning was constrained by MaxMV. For example, if 16x16 was the lowest sad+cost and MaxMV was set to 10, the partitioning was not constrained. However, if 8x8 was the lowest sad+cost and MaxMV was set to 1, partitioning was constrained by MaxMV and this bit would be set.

There are also 3 counter values. One is to report the total number of search units processed by the back-end (max is 48). Another is to report the total time the front-end is starved due to cache misses, counted in divisions of 16 clocks (max is 1024*16 clocks). This will most likely be active at the beginning of a VME request, however, even after processing has begun, if any front-end stalls occur this counter should resume counting. Hence, when the VME request has finished, this counter will have the total time the front-end is stalled. The third field is used to report the total time the back-end consumed for computation, also counted in divisions of 16 clocks (max is 256*16 clocks) [Note: this should include any bubbles in the pipe, simply put, if front-end is not stalled, this counter should be free-running]. Thus, by adding total front-end starved time with total back-end computation time, the exact total VME message time can be obtained.

Changes

VME will retain the grand majority of previous VME features; however a new programming model is required. Additionally, new features have been added. Overall, the engine is more flexible, has more features and is faster than previous generations. See Haswell New Features Overview for further details.

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 3D_Media_GPGPU chapter, Media_GPGPU_Pipeline for more details).

**VME_SEARCH_PATH_LUT_STATE**

Up to eight VME_SEARCH_PATH_LUT_STATE allowed for a message to select. Each state contains one set of search path locations, and four sets of rate distortion cost function LUT for various modes and rate distortion cost function LUT for motion vectors (relative to cost center). Motion vector cost function is provided as a piece-wise-linear curve with only the values of the power-of-2 positions provided.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:13</td>
<td></td>
<td><strong>Search Path</strong></td>
</tr>
<tr>
<td>0</td>
<td>31:24</td>
<td>Search Path Location [3] (X, Y) – Relative distance from location [2]</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>Search Path Location [1] (X, Y) – Relative distance from location [0]</td>
</tr>
<tr>
<td>7:4</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4, (e.g. 0x3 + 0xE = 0x1)</td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td>Search Path Distance [0] (X) – specifies relative X distance of the next walk from the starting position in unit of SU.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td>1:13</td>
<td></td>
<td>Search Path Location [4 – 55] (X, Y)</td>
</tr>
<tr>
<td>14:31</td>
<td></td>
<td>RD LUT SET 0-4</td>
</tr>
<tr>
<td>14</td>
<td>31:24</td>
<td>LUT_MbMode [9] for Set 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>LUT_MbMode [8] for Set 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>LUT_MbMode [9] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>LUT_MbMode [8] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>15</td>
<td>31:24</td>
<td>LUT_MbMode [9] for Set 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>LUT_MbMode [8] for Set 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>LUT_MbMode [9] for Set 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>LUT_MbMode [8] for Set 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>16</td>
<td>31:24</td>
<td>LUT_MbMode [3] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>LUT_MbMode [2] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>LUT_MbMode [1] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>LUT_MbMode [0] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td>17</td>
<td>31:24</td>
<td>LUT_MbMode [7] for Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 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) |
        |      | 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  
24-27  
28-31  

Finish RD LUT SET 1  
Finish RD LUT SET 2  
Finish RD LUT SET 3

The assignment of LUT_MbMode entries is according to the MbTypeEx definition:

<table>
<thead>
<tr>
<th>Index to LUT_MbMode</th>
<th>MbTypeEx</th>
<th>Description</th>
<th>AVC</th>
<th>VC1</th>
<th>MPEG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MODE_INTRA_NONPRED</td>
<td>For INTRA8x8 and INTRA4x4 only. Added per 8x8 for INTRA8x8, and per 4x4 for INTRA4x4</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>1</td>
<td>MODE_INTRA</td>
<td>Added per 16x16 macroblock</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>mode_Intra_16x16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MODE_INTRA_8x8</td>
<td>Added per 16x16 macroblock</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>3</td>
<td>MODE_INTRA_4x4</td>
<td>Added per 16x16 macroblock</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>8</td>
<td>MODE_INTER</td>
<td>Added per 16x16 macroblock</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>mode_Inter_16x16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The value of each byte of the LUTs will be viewed as a pair of 4-bit units: (shift, base), and constructed as base << shift.

For example, an entry 0x4A represents the value (0xA<<0x4) = 10*16 = 160. Encoded value must fit in 12-bits (unsigned number); otherwise, the hardware behavior is undefined.

The only exception is for Index of 9, MODE_INTER_BWD, which is used as a bias for the two search directions. It is a signed number instead, in the form of (SU3U4) = (sign, shift, base). The sign bit indicates whether the bias is added to the forward (if sign = 1) or the backward (if sign = 0). The bias has a magnitude of (base << shift), which has 11-bits precision. It should be noted that the number is always added, there is no subtraction.

Intra Modes only apply to AVC standard. The mode penalty doesn’t apply to Skip Mode Checking. Note that while other mode penalty applies to a fixed macroblock partition, MODE_INTRA_NONPRED applies to all three intra modes. It is a constant cost adder for intra-mode coding regardless of the block size.

For source block that is less than 16x16 (like a 16x8 source block), the proper mode penalty that is stated as added per 16x16 macroblock is added once to the source block (like MODE_INTER_16x8 is added once to a 16x8 source block). It will not be divided by the source block size.

The LUT_MV is added to all motion vector coordinate deltas in quarter-pel unit except for the SKIP mode, which no costing penalty applies. Given motion vector coordinate, e.g. mvx, which is in quarter-pel precision (S5.2), the mv delta is defined to be its difference from the given costing center, e.g. ccx, and the costing penalty is applied to dx = |mvx-ccx|. The cost penalty is a piecewise linear interpolation from the LUT_MV table whereas the values on power-of-2 integer samples are provided. The piecewise linear interpolation is performed using quarter-pel precision, while the LUT_MV are only provided for the given power-of-2 integer positions. The maximum distance provided in the table is 64 pixels. A linear ramp with gradient of 1 on integer distance is applied for bigger distances with maximum penalty capped to 0x3FF (10 bits). Thus,

\[
\text{Costing\_penalty\_x} = \text{LUT\_MV}[\text{int}(dx)], \text{ if } dx < 3 \text{ and } dx = \text{int}(dx);
\]
Costing\_penalty\_x = LUT\_MV[p+1], else if \(dx = 2^p\), for any \(p \leq 6\);

Costing\_penalty\_x = LUT\_MV[p+1] + \((LUT\_MV[p+2] – LUT\_MV[p+1])^k) \gg p\),
else if \(dx = 2^p + k\), for any \(p < 6\) and \(k \leq 2^p\), and

Costing\_penalty\_x = \text{min}(LUT\_MV[7] + \text{int}(dx) – 64, 255), else if \(dx > 64\).

The total costing penalty for a motion vector is

\[\text{Costing\_penalty} = \text{Costing\_penalty\_x} + \text{Costing\_penalty\_y}\]

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 \textbf{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 \((0,0)\).

**Haswell New Features Overview**

Previously issued PRMs list out numerous features of the VME unit. The VME is by default a superset of those features. However, some features are added and some legacy features are removed (where possible). For deeper familiarity with the VMEs for previous chipsets, see their respective PRMs. Discussion of previous VME instantiations only occurs here when required to explain a new or removed feature.

The new features are listed below and descriptions are provided in the sections that follow:

- IME Repartition (Dedicated IME Pipeline)
- HW Accelerated Chroma Intra
- Improved Skip Decision (FTQ)
- HW Assisted Multi-Reference Support
- Chroma Inter Mode
- Increased SAD Precision

**IME Repartition (Dedicated IME Pipeline)**

This is the most significant change to VME. It improves both performance (significantly) and flexibility of the unit, but imposes an increase of complexity to the kernel (other features are added with intent of offsetting this burden). VME will be divided into 2 internal pipelines for increased parallelism: the IME message will be given a new dedicated datapath separate from where the SIC and FBR messages will be performed. This added parallelism will increase VME performance on the order of 1.5x.

Specifically, instead of 1 single call to VME where all 5 subfunctions (Skip, Intra, IME, FME, BME) would be performed serially, the VME has been divided into 3 atomic operations that SW can configure anyway it chooses. The final mode decision is done inside the kernel.

- Skip & Intra operations have been joined together in a \textit{Skip & Intra Check} message (SIC).
- Integer Motion Estimation has been isolated in a standalone IME message (IME).
- FME & BME have been combined into a *Fraction & Bidirectional Refinement* message (FBR).

**Example Kernel S Pipeline**

The preceding figure shows an example SW pipeline to reproduce the HW-managed pipeline of. The following figures show the monolithic and atomic pipelines in more detail.
VME Flowchart

The preceding figure shows the HW-managed VME flow, which allows for sub-functions to be enabled/disabled independently and certain exit points to be activated by satisfying a quality threshold.
The preceding figure shows the managed VME atomic messages, which is similar to VME but split at key points between subfunctions. This is done to isolate IME for performance reasons and provide more flexibility for SW to innovate.
**HW Accelerated Chroma Intra**

The chroma mode decision consumes a significant portion of the kernel footprint and time, moving it to HW will offset some of the cost to support the IME repartition. The kernel will fetch the neighbor MBs CbCr pairs and provide them in the SIC message and VME will evaluate the 4 chroma modes and return the mode with the least combined CbCr distortion. Note: only NV12 is supported. A U4U4 value *Chroma Intra Mode Cost* will be used to penalize the different chroma intra modes. DC gets no penalty, Horz & Vert get 1x penalty, and Plane gets 2x added to the final SAD of each mode before VME selects the winner. (e.g. add cost for each mode: DC=0x, Horz=1x, Vert=1x, Plane=2x). The kernel can select HW intra prediction for Y+CbCr, Y only, or no intra prediction. However, all 4 SIC message phases are expected to be sent by the kernel in any of these cases.

<table>
<thead>
<tr>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
<th>Cb</th>
<th>Cr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
<tr>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
<td>Cb</td>
<td>Cr</td>
</tr>
</tbody>
</table>

**Improved Skip Decision (FTQ)**

The skip decision has been enhanced to include an accurate AVC forward transform for skip estimation. This feature is in addition to the previous SAD or HAAR skip estimation. The results are then compared one coefficient at a time against a user-specified threshold to emulate quantization's zeroing effect. The user is returned the count of coefficients that exceeded their threshold along with a sum of the amount exceeded, both grouped at the 8x8 block level. Coefficients of similar frequencies are grouped together and will share the same threshold and are show in the matrix below. Note: the DC threshold has 16b of precision whereas the remaining thresholds are 8b. Also, only the 4x4 transform is supported.

```
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
```

Note: There is a performance penalty of using this feature as the throughput of SIC is reduced and hence it can be disabled entirely to save performance when necessary. However, the performance loss should not affect app performance.
HW Assisted Multi-Reference Support

VME messages will provide 4b reference identifier (refid) to the HW. VME will keep track of this field and report out the refid along with the rest of the output. For IME, this is most useful when combined with the streamin\streamout feature for multi-call as the references can now be swapped between calls and when the global record is updated, VME will keep track of which reference that shape comes from for the kernel. For the cases of SIC and FBR, this is mostly pass-through, as those messages will not change the partitioning decision from IME.

The Ref ID Cost is defined as follows. There will be two penalty costing modes: AVC and Linear. They are applied to each major shape partition independently. For the AVC costing mode, these RefIDs get the associated penalty x times per major shape partition: 0 => 0x, 1-2 => 1x, 3-6 => 2x, 7-15 => 3x. For the Linear costing mode, the penalty is applied x times per major shape partition where x is equal to the RefID value.
Chroma Inter Mode

In order to improve quality of complex chroma content, which is not common, VME will add *low-cost* support for performing skip, IME, FME, and BME of chroma pixels. Each VME message will be performing
either luma inter or chroma inter skip check, search, or refinement. Performance of chroma is not expected to be at rate with luma as the chroma pixel arrangement does not match with the uArch of the previous pipelines, so some clocks will be wasted when computes are performed illegally (these outputs are ignored, but still computed, hence performance is lost). This is the nature of the low-cost designation. For Chroma Inter Mode SIC messages, no intra prediction (either luma or chroma) will be performed. However, all 4 SIC message phases will be expected to be sent by the kernel. Input and output fields within the PRM will have special interpretation when chroma inter mode is enabled. Specifically, the MB source location, IME reference window location and size, skip center, and cost center input fields and the return motion vectors are to be interpreted as follows:

- All of these fields will be in context of the chroma Cb or Cr surface (e.g. nothing in relation to luma or CbCr pairs).
- This requires the HW to double the x integer component of the source, reference window location, and skip center for working with data on NV12 surfaces.
- Regarding the reference window height, the user can now select multiples of 2 instead of multiples of 4. However, the HW might overfetch what is required to simplify the design to match the luma processing mode (RefH will end up snapping to nearest larger multiple of 4 and not be lesser than 20 tall).
- The skip center and cost center will be S12.3 values. The LSB, which denotes 8\textsuperscript{th} pel precision, will be ignored by the HW (only quarter pel precision is supported) for both x and y components.
- The return MVs will also be S12.3 values. Since the internal HW works on CbCr pairs instead of Y values in this mode, the integer x component of the motion vectors will need to be divided by 2 to denote the Cb motion vector. Also, both the fractional x and y components will need to be multiplied by 2 to artificially produce the 3b fractional precision result.

### Increased SAD Precision

In previous generations VME would perform clamping of the MSB and drop the LSB of each 4x4 SAD\textbackslash HAAR block, and impose saturation checks along the entire shape reduction tree. This was done to save HW cost, but the cost is not significant and there is a marginal risk to quality. Hence, the full SAD precision will be retained throughout the calculation and shape reduction tree. The net effect is the distortion values returned to the kernel will now be 16b instead of 14b.

### Legacy Feature Removal

The IME repartition feature allows the removal of features which are no longer needed and can be emulated by the SW if required. Additionally, defining the new message types allowed for the opportunity to cleanup waste in the messages.

- Early exits
  - All of the performance thresholds (except EarlyIMEStop) are implicitly removed when the atomic message methodology is applied. Now, the kernel will check the results of each stage and determine if continued computation is required. EarlyIMEStop is still supported,
as this allows early termination of the IME searching operation, which the kernel does not have visibility into IME between SUs.

- **FME Repartition**
  - The cost in HW of duplicating the partitioning logic between the IME and CRE pipelines within VME is very large and outweighs the quality improvement.

- **Alternate Candidate**
  - The spirit of this feature can be better implemented by the atomic message types and through multiple FBR calls via the kernel per MB than what was previously achievable.

- **VME State**
  - The addition of a 2nd pipeline that needs access to this cache presents significant performance risk as stalls are more likely. Additionally, the search path and cost lookup tables can be added to the input message payload without significant impact to the kernel or hardware. This will eliminate the need for the IME and CRE pipes to access SVSM for VME state. An additional benefit of this is increased programmability during execution of a frame, as the kernel can now modify the search path and/or cost lookup tables on the fly and are no longer bound to fit within a limited number of locations within the lookup table. Hence, this is a faster and more flexible solution.

**Software Interface – PRM Highlights**

**Message Structure Overview**

- VME is divided into 3 message types: IME (Integer Motion Estimation), SIC (Skip & Intra Check), and FBR (Fractional & Bidirectional Refinement).
- 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 will be required to pack the necessary GRFs together to generate the correct message phase sequence prior to calling VME (i.e. 7 phases for SIC, 5, 7 or 9 phases for IME, and 7 phases for FBR).
- The return data will be structurally common amongst 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 will be returned.
- In total, 19 message phases are based on Gen7 VME and 5 new message phases are defined for Gen7.5 VME (1 Universal Input, 1 SIC Input, 2 IME Input, 1 Output).
- Additionally, the placement of individual fields within the message phases is generally identical to that of previous generations.
### IME & IDM Message Descriptor

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><strong>Header Present.</strong> If set, indicates that the message includes the header. This bit must be 1 for all VME messages. Format = Enable</td>
</tr>
<tr>
<td>18</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>17</td>
<td>Reserved. Format = Enable</td>
</tr>
<tr>
<td>16</td>
<td><strong>Stream-In Enable.</strong> 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 &amp; dual record) and 2 additional phases otherwise. Format = Enable</td>
</tr>
<tr>
<td>15</td>
<td><strong>Stream-Out Enable.</strong> 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 &amp; dual record) and 2 additional phases otherwise. Format = Enable</td>
</tr>
</tbody>
</table>
| 14:13| **Message Type**  
|      | 00: Reserved  
|      | 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] |
### SIC and FBR Message Descriptor

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><strong>Header Present.</strong> If set, indicates that the message includes the header. This bit must be set to one for all VME messages. Format = Enable</td>
</tr>
<tr>
<td>18:15</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>
| 14:13 | **Message Type**  
00: Reserved  
01: SIC  
10: Reserved  
11: FBR |
| 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

<table>
<thead>
<tr>
<th>GRF</th>
<th>Name</th>
<th>Msgs</th>
<th>New</th>
<th>Major Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Uni0</td>
<td>ALL</td>
<td>No</td>
<td>Universal control data</td>
</tr>
<tr>
<td>1</td>
<td>Uni1</td>
<td>ALL</td>
<td>No</td>
<td>Universal control data</td>
</tr>
<tr>
<td>2</td>
<td>Uni2</td>
<td>ALL</td>
<td>Yes</td>
<td>Costs, FT Matrix, FBR Modes</td>
</tr>
<tr>
<td>3</td>
<td>SIC0</td>
<td>SIC</td>
<td>No</td>
<td>Luma intra pix, modes, masks</td>
</tr>
<tr>
<td>4</td>
<td>SIC1</td>
<td>SIC</td>
<td>No</td>
<td>Luma intra pix, modes, masks</td>
</tr>
<tr>
<td>5</td>
<td>SIC2</td>
<td>SIC</td>
<td>No</td>
<td>Costs, FT Matrix, FBR Modes</td>
</tr>
<tr>
<td>6</td>
<td>SIC3</td>
<td>SIC</td>
<td>Yes</td>
<td>Chroma intra pix &amp; masks</td>
</tr>
<tr>
<td>7</td>
<td>IME0</td>
<td>IME</td>
<td>Yes</td>
<td>Search Path</td>
</tr>
<tr>
<td>8</td>
<td>IME1</td>
<td>IME</td>
<td>Yes</td>
<td>Search Path</td>
</tr>
<tr>
<td>9</td>
<td>IME2</td>
<td>IME</td>
<td>No</td>
<td>Streamin/Streamout</td>
</tr>
<tr>
<td>10</td>
<td>IME3</td>
<td>IME</td>
<td>No</td>
<td>Streamin/Streamout</td>
</tr>
<tr>
<td>11</td>
<td>IME4</td>
<td>IME</td>
<td>No</td>
<td>Streamin/Streamout</td>
</tr>
<tr>
<td>12</td>
<td>IME5</td>
<td>IME</td>
<td>No</td>
<td>Streamin/Streamout</td>
</tr>
<tr>
<td>13</td>
<td>FBR0</td>
<td>FBR</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>14</td>
<td>FBR1</td>
<td>FBR</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>15</td>
<td>FBR2</td>
<td>FBR</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>16</td>
<td>FBR3</td>
<td>FBR</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
</tbody>
</table>

## Input Message Phases by Type

VME message types require only a subset of the total GRFs of control data.

<table>
<thead>
<tr>
<th>Phase</th>
<th>SIC</th>
<th>IME</th>
<th>FBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Uni0</td>
<td>Uni0</td>
<td>Uni0</td>
</tr>
<tr>
<td>1</td>
<td>Uni1</td>
<td>Uni1</td>
<td>Uni1</td>
</tr>
<tr>
<td>2</td>
<td>Uni2</td>
<td>Uni2</td>
<td>Uni2</td>
</tr>
<tr>
<td>3</td>
<td>SIC0</td>
<td>IME0</td>
<td>FBR0</td>
</tr>
<tr>
<td>4</td>
<td>SIC1</td>
<td>IME1</td>
<td>FBR1</td>
</tr>
<tr>
<td>5</td>
<td>SIC2</td>
<td>IME2</td>
<td>FBR2</td>
</tr>
<tr>
<td>6</td>
<td>SIC3</td>
<td>IME3</td>
<td>FBR3</td>
</tr>
<tr>
<td>7</td>
<td>IME4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IME5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Output GRFs

<table>
<thead>
<tr>
<th>GRF</th>
<th>Name</th>
<th>Msgs</th>
<th>New</th>
<th>Major Contents</th>
</tr>
</thead>
</table>

383
<table>
<thead>
<tr>
<th>GRF</th>
<th>Name</th>
<th>Msgs</th>
<th>New</th>
<th>Major Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ret0</td>
<td>ALL</td>
<td>No</td>
<td>Best MB Control Data</td>
</tr>
<tr>
<td>1</td>
<td>Ret1</td>
<td>ALL</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>2</td>
<td>Ret2</td>
<td>ALL</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>3</td>
<td>Ret3</td>
<td>ALL</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>4</td>
<td>Ret4</td>
<td>ALL</td>
<td>No</td>
<td>8 Inter 4x4 MVs</td>
</tr>
<tr>
<td>5</td>
<td>Ret5</td>
<td>ALL</td>
<td>No</td>
<td>Inter Block Distortions</td>
</tr>
<tr>
<td>6</td>
<td>Ret6</td>
<td>ALL</td>
<td>Yes</td>
<td>Block Ref Indices &amp; FTQ Data</td>
</tr>
<tr>
<td>7</td>
<td>IME2</td>
<td>IME</td>
<td>No</td>
<td>Streamin\Streamout</td>
</tr>
<tr>
<td>8</td>
<td>IME3</td>
<td>IME</td>
<td>No</td>
<td>Streamin\Streamout</td>
</tr>
<tr>
<td>9</td>
<td>IME4</td>
<td>IME</td>
<td>No</td>
<td>Streamin\Streamout</td>
</tr>
<tr>
<td>10</td>
<td>IME5</td>
<td>IME</td>
<td>No</td>
<td>Streamin\Streamout</td>
</tr>
</tbody>
</table>

**Output Message Phases by Type**

All message types return 7 phases. 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.

<table>
<thead>
<tr>
<th>Phase</th>
<th>SIC</th>
<th>IME</th>
<th>FBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ret0</td>
<td>Ret0</td>
<td>Ret0</td>
</tr>
<tr>
<td>1</td>
<td>Ret1</td>
<td>Ret1</td>
<td>Ret1</td>
</tr>
<tr>
<td>2</td>
<td>Ret2</td>
<td>Ret2</td>
<td>Ret2</td>
</tr>
<tr>
<td>3</td>
<td>Ret3</td>
<td>Ret3</td>
<td>Ret3</td>
</tr>
<tr>
<td>4</td>
<td>Ret4</td>
<td>Ret4</td>
<td>Ret4</td>
</tr>
<tr>
<td>5</td>
<td>Ret5</td>
<td>Ret5</td>
<td>Ret5</td>
</tr>
<tr>
<td>6</td>
<td>Ret6</td>
<td>Ret6</td>
<td>Ret6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>IME2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>IME3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>IME4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>IME5</td>
<td></td>
</tr>
</tbody>
</table>
## 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

<table>
<thead>
<tr>
<th>BTI</th>
<th>Direction</th>
<th>Number</th>
<th>Field</th>
<th>Universal Input M1.6 RefIDs (4b Value per Block)</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>FWD</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>BWD</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>FWD</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>BWD</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>FWD</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>6</td>
<td>BWD</td>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>FWD</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>BWD</td>
<td>3</td>
<td>N/A</td>
<td>N/A</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>FWD</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
</tr>
<tr>
<td>10</td>
<td>BWD</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>FWD</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>BTI</td>
<td>Direction</td>
<td>Number</td>
<td>Field</td>
<td>FWD 0</td>
<td>BWD 0</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>12</td>
<td>BWD</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>13</td>
<td>FWD</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>14</td>
<td>BWD</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>15</td>
<td>FWD</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
</tr>
<tr>
<td>16</td>
<td>BWD</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
</tr>
<tr>
<td>17</td>
<td>FWD</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>18</td>
<td>BWD</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>19</td>
<td>FWD</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
</tr>
<tr>
<td>20</td>
<td>BWD</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
</tr>
<tr>
<td>21</td>
<td>FWD</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
</tr>
<tr>
<td>22</td>
<td>BWD</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
</tr>
<tr>
<td>23</td>
<td>FWD</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
</tr>
<tr>
<td>24</td>
<td>BWD</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
</tr>
<tr>
<td>25</td>
<td>FWD</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Universal Input M1.6 RefIDs (4b Value per Block)

<table>
<thead>
<tr>
<th>BTI</th>
<th>Direction</th>
<th>Number</th>
<th>Field</th>
<th>FWD 0</th>
<th>BWD 0</th>
<th>FWD 1</th>
<th>BWD 1</th>
<th>FWD 2</th>
<th>BWD 2</th>
<th>FWD 3</th>
<th>BWD 3</th>
<th>BTI Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>BWD</td>
<td>12</td>
<td>N/A</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>27</td>
<td>FWD</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>28</td>
<td>BWD</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>29</td>
<td>FWD</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>30</td>
<td>BWD</td>
<td>14</td>
<td>N/A</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>31</td>
<td>FWD</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>32</td>
<td>BWD</td>
<td>15</td>
<td>N/A</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>N/A</td>
<td>15</td>
<td>= RefID * 2 + 2</td>
</tr>
</tbody>
</table>

### Interlaced Content

<table>
<thead>
<tr>
<th>BTI</th>
<th>Direction</th>
<th>Number</th>
<th>Field</th>
<th>FWD 0</th>
<th>BWD 0</th>
<th>FWD 1</th>
<th>BWD 1</th>
<th>FWD 2</th>
<th>BWD 2</th>
<th>FWD 3</th>
<th>BWD 3</th>
<th>BTI Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>= From input</td>
</tr>
<tr>
<td>1</td>
<td>FWD</td>
<td>0</td>
<td>Top</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>2</td>
<td>BWD</td>
<td>0</td>
<td>Top</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>3</td>
<td>FWD</td>
<td>0</td>
<td>Bot</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>BTI</td>
<td>Direction</td>
<td>Number</td>
<td>Field</td>
<td>FWD 0</td>
<td>BWD 0</td>
<td>FWD 1</td>
<td>BWD 1</td>
<td>FWD 2</td>
<td>BWD 2</td>
<td>FWD 3</td>
<td>BWD 3</td>
<td>BTI Equation</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>---------------</td>
</tr>
<tr>
<td>4</td>
<td>BWD</td>
<td>0</td>
<td>Bot</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>5</td>
<td>FWD</td>
<td>1</td>
<td>Top</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>6</td>
<td>BWD</td>
<td>1</td>
<td>Top</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
<td>2</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>7</td>
<td>FWD</td>
<td>1</td>
<td>Bot</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>8</td>
<td>BWD</td>
<td>1</td>
<td>Bot</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>9</td>
<td>FWD</td>
<td>2</td>
<td>Top</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>10</td>
<td>BWD</td>
<td>2</td>
<td>Top</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>11</td>
<td>FWD</td>
<td>2</td>
<td>Bot</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>12</td>
<td>BWD</td>
<td>2</td>
<td>Bot</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>13</td>
<td>FWD</td>
<td>3</td>
<td>Top</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>14</td>
<td>BWD</td>
<td>3</td>
<td>Top</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>N/A</td>
<td>6</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>15</td>
<td>FWD</td>
<td>3</td>
<td>Bot</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>= RefID * 2 + 1</td>
</tr>
<tr>
<td>16</td>
<td>BWD</td>
<td>3</td>
<td>Bot</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>= RefID * 2 + 2</td>
</tr>
<tr>
<td>17</td>
<td>FWD</td>
<td>4</td>
<td>Top</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>= RefID * 2</td>
</tr>
<tr>
<td>BTI</td>
<td>Direction</td>
<td>Number</td>
<td>Field</td>
<td>FWD 0</td>
<td>BWD 0</td>
<td>FWD 1</td>
<td>BWD 1</td>
<td>FWD 2</td>
<td>BWD 2</td>
<td>FWD 3</td>
<td>BWD 3</td>
<td>BTI Equation</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td>18</td>
<td>BWD</td>
<td>4</td>
<td>Top</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>19</td>
<td>FWD</td>
<td>4</td>
<td>Bot</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>20</td>
<td>BWD</td>
<td>4</td>
<td>Bot</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>N/A</td>
<td>9</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>21</td>
<td>FWD</td>
<td>5</td>
<td>Top</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>22</td>
<td>BWD</td>
<td>5</td>
<td>Top</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>23</td>
<td>FWD</td>
<td>5</td>
<td>Bot</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>24</td>
<td>BWD</td>
<td>5</td>
<td>Bot</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>25</td>
<td>FWD</td>
<td>6</td>
<td>Top</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>26</td>
<td>BWD</td>
<td>6</td>
<td>Top</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>N/A</td>
<td>12</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>27</td>
<td>FWD</td>
<td>6</td>
<td>Bot</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>28</td>
<td>BWD</td>
<td>6</td>
<td>Bot</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>13</td>
<td>= RefID * 2  + 2</td>
</tr>
<tr>
<td>29</td>
<td>FWD</td>
<td>7</td>
<td>Top</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>= RefID * 2  + 1</td>
</tr>
<tr>
<td>30</td>
<td>BWD</td>
<td>7</td>
<td>Top</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>N/A</td>
<td>14</td>
<td>= RefID * 2  + 2</td>
</tr>
</tbody>
</table>
Glossary of Messages

This section describes the glossary of messages in regard to Media Sampler.

Programming Note:

- Use of any messages to the Video Motion Estimation function while there are any messages to any sampler function is not allowed.

Universal Input Message Phases

Major changes:

- 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 will ignore these fields under messages where that field is invalid. Hardware output for non valid fields is undefined.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.7</td>
<td>31:0</td>
<td>Reference Region Height (RefHeight)</td>
</tr>
<tr>
<td>M0.6</td>
<td>31:0</td>
<td>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.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ValidMsgType = IME</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = U8</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range = [8, 64]</strong></td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td><strong>Reference Region Width (RefWidth)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Reference Region Width (RefWidth):</strong> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ValidMsgType = IME</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = U8</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range = [20, 128]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Please make sure the reference windows are not completely outside of the video frame. In that case, VME behavior is undefined.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Reference Window size must be &lt;= Surface Size, otherwise VME behavior is undefined.</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>Dispatch ID.</strong> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ValidMsgType = SIC, IME, FBR</strong></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:0</td>
<td>Ignored (reserved for hardware delivery of binding table pointer)</td>
</tr>
<tr>
<td>M0.3</td>
<td>31</td>
<td>2D 7 by 7 Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td>30:24</td>
<td></td>
<td><strong>Sub-Macroblock Sub-Partition Mask (SubMbPartMask)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Sub-Macroblock Sub-Partition Mask (SubMbPartMask):</strong> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxxxxx1: 16x16 sub-macroblock disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxxxx1x: 2x(16x8) sub-macroblock within 16x16 disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx1xx: 2x(8x16) sub-macroblock within 16x16 disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxx1xxx: 1x(8x8) sub-partition for 4x(8x8) within 16x16 disabled</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>xx1xxxx: x2(8x4) sub-partition for 4x(8x8) within 16x16 disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x1xxxxx: x2(4x8) sub-partition for 4x(8x8) within 16x16 disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1xxxxxxx: 4x(4x4) sub-partition for 4x(8x8) within 16x16 disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111111: Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: Invalid to have all partitions disabled in the IME call.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ValidMsgType = IME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Usage note: One example usage of only enabling 4x(4x4) sub-partition while all other partitions are disabled is for video processing, whereas parallel motion searches are performed for 16 4x4 blocks. For that no further block combination (into larger sub-partitions/sub-macroblocks) is needed.</td>
<td></td>
</tr>
<tr>
<td>23:22</td>
<td>Intra SAD Measure Adjustment (IntraSAD)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00b: none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01b: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10b: Haar transform adjusted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11b: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ValidMsgType = SIC</td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Inter SAD Measure Adjustment (InterSAD)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00b: none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01b: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10b: Haar transform adjusted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11b: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ValidMsgType = SIC, IME, FBR, IDM</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ValidMsgType = SIC</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td><strong>BME disable for FBR Message (BMEDisableFBR)</strong>&lt;br&gt;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).&lt;br&gt;0 = BME enabled&lt;br&gt;1 = BME disabled&lt;br&gt;ValidMsgType = FBR</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td><strong>Forward Transform Skip Check Enable (FTEnable)</strong>&lt;br&gt;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.&lt;br&gt;0 = FT disabled&lt;br&gt;1 = FT enabled&lt;br&gt;ValidMsgType = SIC</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td><strong>Process Inter Chroma Pixels Mode (InterChroaZmaMode)</strong>&lt;br&gt;This bit switches the inter operations from luma mode to chroma mode.&lt;br&gt;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.&lt;br&gt;MBMode is always 8x8.&lt;br&gt;MBSUBShape is either 8x8 or 4x4 indicated by LSB[1:0]. Bits[7:2] are MBZ.&lt;br&gt;For MBSUBShape of 4x4, SubPredMode is mapped to each 4x4 shape.&lt;br&gt;Only 8x8 and 4x4 ModeCost are valid.&lt;br&gt;Source block size is ignored.&lt;br&gt;Streamin\streamout distortions are overloaded on 16x16 (Chroma8x8) and 8x8 (Chroma4x4).&lt;br&gt;BilinearEnable is ignored (Chroma can only perform bilinear filtering)&lt;br&gt;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).&lt;br&gt;ValidMsgType = SIC, IME, FBR</td>
</tr>
</tbody>
</table>
| 15    |      | **Disable Field Cache Allocation**<br>This field, when set to 1, disables the optimized field cache line method in the Sampler Cache for
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>reference block data when RefAccess is 1 (field based). It is ignored by hardware if RefAccess is 0. 0 – frame or field cache lines according to RefAccess 1 – always frame cache lines</td>
</tr>
</tbody>
</table>
| 14    |      | **Skip Mode Type**  
For B_DIRCET_16x16, both motion vectors of the skip center pair 0 are used.  
For B_DIRCET_8x8s, all four skip center pairs are **fully** used (VME will never try to combine them with non-skip shapes from IME, FME, or BME).  
**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 will be used)  
Note: SkipModeType should be programmed to 1MVP for non-16x16 Source size.  
ValidMsgType = SIC |
| 13:12 |      | **Sub-Pel Mode (SubPelMode)**  
This field defines the half/quarter pel modes. The mode is inclusive, i.e., 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>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000b</td>
<td><strong>Code</strong>&lt;br&gt;Single reference, single record and single start.&lt;br&gt;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.&lt;br&gt;This is the common single directional motion search mode.</td>
</tr>
<tr>
<td></td>
<td>001b</td>
<td><strong>Code</strong>&lt;br&gt;Single reference, single record and dual start.&lt;br&gt;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.&lt;br&gt;This is used for single direction adaptive search.</td>
</tr>
<tr>
<td></td>
<td>011b</td>
<td><strong>Code</strong>&lt;br&gt;Single reference, dual record (and implied dual start).&lt;br&gt;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.&lt;br&gt;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.&lt;br&gt;This may be used for search for multiple motion search candidates/predicators.</td>
</tr>
<tr>
<td></td>
<td>111b</td>
<td><strong>Code</strong>&lt;br&gt;Dual reference, dual record (and implied dual start).&lt;br&gt;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.&lt;br&gt;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.&lt;br&gt;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.&lt;br&gt;If BiSubMbPartMask is set to 1111'b, bidirectional search is disabled. VME will output only the best unidirectional search results. Otherwise, BME will be performed.&lt;br&gt;Note that bidirectional search and sub-pel refinement are orthogonal features that can be enabled independently.</td>
</tr>
</tbody>
</table>

**Reference Access (RefAccess)**

This field defines how the reference blocks are accessed from the reference frames. It indicates if the source picture is a frame picture or a field picture.

*Programming Note: For all known video coding standards, reference pictures always have the same picture type as the source picture. Therefore, this field should be programmed to be the same as*
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>SrcAccess</strong>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: frame based</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: field based</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td><strong>Source Access (SrcAccess)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: frame based</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: field based</td>
</tr>
<tr>
<td>5:4</td>
<td></td>
<td><strong>Inter MbType Remap (MbTypeRemap):</strong> 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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00b: no remapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11b: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = IME, FBR</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The following text needs to be maintained so that we can bring back the feature in the next opportunity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Will be used for Field 8x8 Enabled:</strong> This field enables 8x8 interlaced–block partitioning (used for VC-1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Enabling Field 8x8 prevents use of subpartitions types 4x4, 4x8 and 8x4, <strong>RefAccess</strong> and <strong>SrcAccess</strong> must be 0 and <strong>SrcSize</strong> must be 16x16 (00). <strong>Field8x8</strong> and <strong>Field16x8</strong> are mutually exclusive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The following text needs to be maintained so that we can bring back the feature in the next opportunity.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|       |      | **Will be used for Field 16x8 Enabled:** This field enables 16x8 interlaced–block partitioning for MPEG-2.  
**Note:** Enabling Field 16x8 prevents use of subpartitions types 8x16, 4x4, 4x8 and 8x4, **RefAccess** and **SrcAccess** must be 0 and **SrcSize** must be 16x16 (00). **Field8x8** and **Field16x8** are mutually exclusive. |
| 1:0   |      | **Source Block Size (SrcSize)**  
This field defines how the 16x16 source block is formed. When Source Block Size is less than 16x16, SU larger than 4x4 will be used.  
00b: 16x16  
01b: 16x8  
10b: Reserved (for 8x16)  
11b: 8x8  
**ValidMsgType** = SIC, IME, FBR |
| M0.2  | 31:16| **Source Y (SrcY)**  
This field defines the vertical position (of the block’s upper-left pixel) in units of pixels for the source block in the source frame.  
The resulting Y address in the reference picture must be in even line aligned within the reference picture. Specifically, if the reference picture is a frame picture, the resulting Y address must be 2-line aligned; if the reference picture is a field picture within a frame storage, and the resulting Y address must be 2-line aligned within the field. I.e. it must be an even number for the frame case, and must be equal to 0 or 1 modulo 4 for the field case. |
| 15:0  |      | **Source X (SrcX)**  
This field defines the horizontal position (of the block’s upper-left pixel) in units of pixels for the source block in the source picture.  
The source block must be within the source picture starting at any integer grid.  
For SIC messages where Intra Compute Type is set to 00 (Luma + Chroma enabled), SrcX must be a multiple of 2. |
| M0.1  | 31:16| **Reference 1 Y Delta (Ref1Y)**  
This field defines the vertical position (of the upper-left corner of the reference region) in units of pixels for Reference 1 region relative to the surface origin. The resulting Y address in the reference picture must be in even line aligned within the reference picture. Specifically, if the reference picture is a frame picture, the resulting Y address must be 2-line aligned; if the reference picture is a field picture within a frame storage, and the resulting Y address must be 2-line aligned within the field. I.e. it must be an even number for the frame case, and must be equal to 0 or 1 modulo 4 for the |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>field case.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> For search control=3, this must equal Ref0Y.</td>
</tr>
<tr>
<td>15:0</td>
<td>54x698 to 126x754</td>
<td><strong>Reference 1 X Delta (Ref1X)</strong>&lt;br&gt;This field defines the horizontal position (of the upper-left corner of the reference region) in units of pixels for Reference 1 region relative to the surface origin.&lt;br&gt;The resulting reference region is allowed to be outside the picture. Pixel replication is applied to generate out of bound reference pixels.&lt;br&gt;This field is only valid when dual reference mode is selected.&lt;br&gt;<strong>Note:</strong> For search control=3, this must equal Ref0X.</td>
</tr>
<tr>
<td></td>
<td>123x667</td>
<td>ValidMsgType = IME &lt;br&gt;Format = S15 &lt;br&gt;Hardware Range: [-2048 to 2047]</td>
</tr>
<tr>
<td>M0.0</td>
<td>31:16</td>
<td><strong>Reference 0 Y Delta (Ref0Y)</strong>&lt;br&gt;This field defines the vertical position (of the upper-left corner of the reference region) in units of pixels for Reference 0 region relative to the surface origin.&lt;br&gt;[HSW]:&lt;br&gt;The resulting Y address in the reference picture must be in even line aligned within the reference picture. Specifically, if the reference picture is a frame picture, the resulting Y address must be 2-line aligned; if the reference picture is a field picture within a frame storage, and the resulting Y address must be 2-line aligned within the field. i.e. it must be an even number for the frame case, and must be equal to 0 or 1 modulo 4 for the field case.</td>
</tr>
<tr>
<td>15:0</td>
<td>60x428</td>
<td><strong>Reference 0 X Delta (Ref0X)</strong>&lt;br&gt;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.&lt;br&gt;The resulting reference region is allowed to be outside the picture. Pixel replication is applied to generate out of bound reference pixels.</td>
</tr>
<tr>
<td>M1.7</td>
<td>31:24</td>
<td><strong>Skip Center Enable Mask (SkipCenterMask):</strong>&lt;br&gt;[bit 31...24]&lt;br&gt;xxxx xxx1: Ref0 Skip Center 0 is enabled [corresponds to M2.0]&lt;br&gt;xxxx xx1x: Ref1 Skip Center 0 is enabled [corresponds to M2.1]&lt;br&gt;xxxx x1xx: Ref0 Skip Center 1 is enabled [corresponds to M2.2]&lt;br&gt;xxxx 1xxx: Ref1 Skip Center 1 is enabled [corresponds to M2.3]</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxx1 xxxx: Ref0 Skip Center 2 is enabled [corresponds to M2.4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xx1x xxxx: Ref1 Skip Center 2 is enabled [corresponds to M2.5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x1xx xxxx: Ref0 Skip Center 3 is enabled [corresponds to M2.6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1xxx xxxx: Ref1 Skip Center 3 is enabled [corresponds to M2.7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Illegal cases:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable both Ref0 and Ref1 Skip Center 0 in case of Skip_1MVP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable both Ref0 and Ref1 for any Skip Center pair in case of Skip_4MVP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC ValidMsgType = SIC</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>IDM Shape Mode Select (IDMShapeMode): [Also see M1.1 bits 30 and 31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HSW]:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Notes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only ref window size of 32x32 (shape16x16), 24x24(shape8x8), 128x16 (shape16x16), and 32x16 (shape16x16) are supported. Search control[2:0] will be default to single ref and single start. Luma only.</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>RefID Cost Mode Select (RefIDCostMode)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selects the RefID costing mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Mode0 (AVC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Mode1 (linear)</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>Enable AC-Only HAAR (AConlyHAAR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit zeros out the DC component in the HAAR SATD block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = AC+DC HAAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = AC HAAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Enable Weighted-SAD\HAAR (WeightedSADHAAR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit enables fixed weighted SAD\HAAR pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Right-shift 4x4 SAD\HAAR for the sub-blocks mapped onto the 16x16 source macroblock by the following amounts:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 1 1 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 1 0 0 1 | 2 1 1 2 | 0 = Flat weighting  
1 = Enable Weighted-SAD\HAAR  
Only supported for source-type luma 16x16. Streamin\Streamout must be disabled.  
ValidMsgType = IME |
| 19 | Source Field Polarity Select (SrcFieldPolarity) | If SrcAccess = 1 (M0.3-6), meaning field based, than the hardware requires this value is 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 that will be fetched from the memory will be:  
SrcY* 2 + SrcFieldPolarity  
Else, this field is ignored by the hardware. |
| 18 | Bilinear Filter Enable (BilinearEnable) | If set, the fractional filter will implement a simple bilinear interpolation filter instead of the 4-tap filter. Note: this is supported for both hpel and qpel interpolation.  
ValidMsgType = SIC, FBR  
Format = Enable |
| 17:16 | MV Cost Scaling Factor (MVCostScaleFactor) | This term allows the user to redefine the precision of the lookup into the LUT_MV based on the MV cost difference from the cost center. The piecewise linear cost function is defined from 0 to 64 in powers of 2 intervals, and the precision of the difference is set by this field. There are 4 precision choices:  
00b: qpel [Qpel difference between MV and cost center: eff cost range 0-15pel]  
01b: hpel [Hpel difference between MV and cost center: eff cost range 0-31pel]  
10b: pel [Pel difference between MV and cost center: eff cost range 0-63pel]  
11b: 2pel [2Pel difference between MV and cost center: eff cost range 0-127pel] |
| 15:8 | Macroblock Intra Structure (MbIntraStruct) | This is a bitmask that specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.  
Note: user must set Bit6=Bit5. |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>Reserved: MBZ (for IntraPredAvailFlagF – F (pixel[-1,7] available for MbAff))</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Reserved: MBZ (for IntraPredAvailFlagA/E – A (left neighbor top half for MbAff))</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>IntraPredAvailFlagE/A – A (Left neighbor or Left bottom half)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>IntraPredAvailFlagB – B (Upper neighbor)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>IntraPredAvailFlagC – C (Upper left neighbor)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>IntraPredAvailFlagD – D (Upper right neighbor)</td>
</tr>
<tr>
<td>1:0</td>
<td></td>
<td>Reserved: MBZ (ChromaIntraPredMode)</td>
</tr>
</tbody>
</table>

ValidMsgType = SIC

**7** Luma Intra Source Corner Swap (IntraCornerSwap): This field specifies the format of the intra luma neighbor pixel format in the message.
0: top neighbors are in sequential order
1: Left-top corner is swapped with the last left-edge neighbor
ValidMsgType = SIC

**6** Non Skip MB Mode Cost Added (NonSkipModeAdded)
This field indicates that the distortion of the survived motion vectors will become non-skip, and the MB mode cost will be added to its distortion.
ValidMsgType = SIC

**5** Non Skip Zero MV Cost Added (NonSkipZMvAdded)
This field indicates that the distortion of the survived motion vectors will become non-skip, and the zero MV component costs will be added to its distortion.
ValidMsgType = SIC

**4:0** Luma Intra Partition Mask (IntraPartMask)
This field specifies which Luma Intra partition is enabled/disabled for intra mode decision.
xxxx1: luma_intra_16x16 disabled
xxx1x: luma_intra_8x8 disabled
xx1xx: luma_intra_4x4 disabled
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Note: for SIC message with IntraComputeType == 00 or 01, at least 1 partition must be enabled. Bits [4:3] MBZ ValidMsgType = SIC</td>
</tr>
<tr>
<td>M1.6</td>
<td>31:28</td>
<td><strong>Bwd Block 3 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td><strong>Fwd Block 3 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td><strong>Bwd Block 2 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Fwd Block 2 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td><strong>Bwd Block 1 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td><strong>Fwd Block 1 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td><strong>Bwd Block 0 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td><strong>Fwd Block 0 RefID</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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. The following restriction is for [HSW] only: For field content, the top fields must have bit 0 as 0 and bottom fields must have bit 0 as 1. RefIDs are programmed similar to SubPredMode: MbMode 16x16: RefID0 MbMode 16x8: RefID0 Top\RefID1 Bottom MbMode 8x16: RefID0 Left\RefID1 Right MbMode 8x8: RefID 0,1,2,3 mapped to 8x8 block number. Performance note: For cases when not MbMode 8x8 and all major shapes share the same reference ID, SW should copy the RefID value into all 4 blocks for surface state fetching optimization.</td>
</tr>
<tr>
<td>M1.5</td>
<td>31:16</td>
<td><strong>Cost Center 1 Delta Y (CostCenter0Y)</strong> This field defines the Y value for the second cost center (associated with the second start) relative to the picture source MB Y value. For FBR messages, Cost Center 1 is used for all backward shapes. ValidMsgType = SIC, IME, FBR Format = S13.2 (2’s comp) Hardware Range: [-512.00 to 511.75]</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Cost Center 1 Delta X (CostCenter1X)</strong> This field defines the X value for the second cost center (associated with the second start) relative to the picture source MB X value.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For FBR messages, Cost Center 1 is used for all backward shapes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S13.2 (2’s comp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware Range: [-2048.00 to 2047.75]</td>
</tr>
<tr>
<td>M1.4</td>
<td>31:16</td>
<td><strong>Cost Center 0 Delta Y (CostCenter0Y):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field defines the Y value for the first cost center (associated with the first start) relative to the picture source MB Y value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For FBR messages, Cost Center 0 is used for all forward shapes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S13.2 (2’s comp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware Range: [-512.00 to 511.75]</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Cost Center 0 Delta X (CostCenter0X):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field defines the X value for the first cost center (associated with the first start) relative to the picture source MB X value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For FBR messages, Cost Center 0 is used for all forward shapes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S13.2 (2’s comp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware Range: [-2048.00 to 2047.75]</td>
</tr>
<tr>
<td>M1.3</td>
<td>31:30</td>
<td><strong>Weighted SAD Control Sub-block 15 (F)</strong></td>
</tr>
<tr>
<td></td>
<td>29:28</td>
<td><strong>Weighted SAD Control Sub-block 14 (E)</strong></td>
</tr>
<tr>
<td></td>
<td>27:26</td>
<td><strong>Weighted SAD Control Sub-block 13 (D)</strong></td>
</tr>
<tr>
<td></td>
<td>25:24</td>
<td><strong>Weighted SAD Control Sub-block 12 (C)</strong></td>
</tr>
<tr>
<td></td>
<td>23:22</td>
<td><strong>Weighted SAD Control Sub-block 11 (B)</strong></td>
</tr>
<tr>
<td></td>
<td>21:20</td>
<td><strong>Weighted SAD Control Sub-block 10 (A)</strong></td>
</tr>
<tr>
<td></td>
<td>19:18</td>
<td><strong>Weighted SAD Control Sub-block 9</strong></td>
</tr>
<tr>
<td></td>
<td>17:16</td>
<td><strong>Weighted SAD Control Sub-block 8</strong></td>
</tr>
<tr>
<td></td>
<td>15:14</td>
<td><strong>Weighted SAD Control Sub-block 7</strong></td>
</tr>
<tr>
<td></td>
<td>13:12</td>
<td><strong>Weighted SAD Control Sub-block 6</strong></td>
</tr>
<tr>
<td></td>
<td>11:10</td>
<td><strong>Weighted SAD Control Sub-block 5</strong></td>
</tr>
<tr>
<td></td>
<td>9:8</td>
<td><strong>Weighted SAD Control Sub-block 4</strong></td>
</tr>
<tr>
<td></td>
<td>7:6</td>
<td><strong>Weighted SAD Control Sub-block 3</strong></td>
</tr>
<tr>
<td></td>
<td>5:4</td>
<td><strong>Weighted SAD Control Sub-block 2</strong></td>
</tr>
<tr>
<td></td>
<td>3:2</td>
<td><strong>Weighted SAD Control Sub-block 1</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 1:0   |      | **Weighted SAD Control Sub-block 0**  
M1.3 31:0 Reserved MBZ. |
| M1.2  | 31:28| **Start Center 1 Y (Start1Y)**  
This field defines the Y position of Search Path 1 relative to the reference Y location. It is in units of SU.  
ValidMsgType = IME  
Format = U4 |
|       | 27:24| **Start Center 1 (Start1X)**  
This field defines the X position of Search Path 1 relative to the reference X location. It is in units of SU.  
The corresponding reference block must be fully within the reference region.  
ValidMsgType = IME  
Format = U4 |
|       | 23:20| **Start Center 0 Y (Start0Y)**  
This field defines the Y position of Search Path 1 relative to the reference Y location. It is in units of SU.  
ValidMsgType = IME  
Format = U4 |
|       | 19:16| **Start Center 0 X (Start0X)**  
This field defines the X position of Search Path 1 relative to the reference X location. It is in units of SU.  
The corresponding reference block must be fully within the reference region.  
ValidMsgType = IME  
Format = U4 |
|       | 15:8 | **Maximum Search Path Length (MaxNumSU)**  
This field defines the maximum number of SUs per reference including the predetermined SUs and the adaptively generated SUs.  
**Note:** Every SU in fixed path will be counted (including the out-bound ones and repeated ones), and in addition for adaptive SUs only the ones actually searched will be added.  
ValidMsgType = IME |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Max Fixed Search Path Length (LenSP)</strong>&lt;br&gt;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.&lt;br&gt;&lt;br&gt;<strong>Note:</strong> Every SU in fixed path will be counted (including the out-bound ones and repeated ones).&lt;br&gt;ValidMsgType = IME&lt;br&gt;Format = U8, with valid range of [1,63]</td>
</tr>
<tr>
<td></td>
<td>M1.1</td>
<td><strong>IDM Shape Mode Select (IDMShapeMode) Extension</strong>&lt;br&gt;[HSW]: Reserved MBZ.</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td><strong>IDM Shape Mode Select (IDMShapeMode) Extension</strong>&lt;br&gt;Reserved MBZ.</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td><strong>Ref pixel bias enable</strong></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td><strong>Unidirectional Mix Disable (UniMixDisable):</strong> 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.)&lt;br&gt;Programmers note: for the case when BMEdisableFBR is set, only the input subpredmode direction will be refined. If BMEdisableFBR is not set, both directions undergo fractional refinement prior to bidirectional refinement, but the subpredmode output will never invert directions if the refinement yielded a better result (subpredmode could change to bidirectional in this case though).&lt;br&gt;This field is MBZ except for cases of Search Control = 111'b (e.g. 7, dual reference).&lt;br&gt;ValidMsgType = IME</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td><strong>Bidirectional Weight (BiWeight)</strong>&lt;br&gt;This field defines the weighting for the backward and forward terms to generate the bidirectional term. This field is only valid for bidirectional search (<strong>SearchCtrl</strong> = 111).&lt;br&gt;ValidMsgType = SIC, FBR&lt;br&gt;Format = U6</td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td><strong>Reserved:</strong> MBZ</td>
</tr>
<tr>
<td></td>
<td>23:22</td>
<td><strong>Reserved:</strong> MBZ</td>
</tr>
<tr>
<td></td>
<td>21:16</td>
<td><strong>Bidirectional Weight (BiWeight)</strong>&lt;br&gt;This field defines the weighting for the backward and forward terms to generate the bidirectional term. This field is only valid for bidirectional search (<strong>SearchCtrl</strong> = 111).&lt;br&gt;ValidMsgType = SIC, FBR&lt;br&gt;Format = U6</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid Values: [16, 21, 32, 43, 48]</td>
</tr>
</tbody>
</table>
| 15:8  |      | **RefId Polarity Bits**  
|       |      | Bit15->bwd block3  
|       |      | Bit14->bwd block2  
|       |      | Bit13->bwd block1  
|       |      | Bit12->bwd block0  
|       |      | Bit11->fwd block3  
|       |      | Bit10->fwd block2  
|       |      | Bit9->fwd block1  
|       |      | Bit8->fwd block0  |
| 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 will return the best partition with MvQuantity not exceeding MaxNumMVs. MaxNumMVs = 0 will only allow 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 restriction 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 will be performed within the IME unit.  
<p>|       |      | This field only takes effect if EarlyImeSuccessEn is set.  |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> 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)</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td><strong>Reserved:</strong> MBZ</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td><strong>Reserved:</strong> MBZ</td>
</tr>
</tbody>
</table>
| 7     |      | **Transform 8x8 Flag For Inter Enable (T8x8FlagForInterEn)**  
|       |      | This field specifies whether Transform8x8Flag is updated for inter mode according 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.  
|       |      | ValidMsgType = IME, IDM |
| 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  
<p>|       |      | ValidMsgType = IME |
| 4:3   |      | <strong>Reserved:</strong> MBZ |
| 2     |      | <strong>Bidirectional Mix Disable (BiMixDis):</strong> 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). |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Usage Example: MPEG2 bidirectional decision is at whole macroblock level, while AVC decision is at subblock level.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: bidirectional decision on subblock level that bidirectional mode is enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: bidirectional decision on whole macroblock</td>
<td></td>
</tr>
<tr>
<td>ValidMsgType = FBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: This must be disabled for SubMbShape with any minors (8x4/4x8/4x4) in the MB.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>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 &gt;=2 or from a dual-start.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: disable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: enable</td>
<td></td>
</tr>
<tr>
<td>ValidMsgType = IME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: disable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1: enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>SIC Forward Transform Coeff Threshold Matrix[5]</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>SIC Forward Transform Coeff Threshold Matrix[4]</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>SIC Forward Transform Coeff Threshold Matrix[3]</td>
</tr>
<tr>
<td>M2.6</td>
<td>31:24</td>
<td>SIC Forward Transform Coeff Threshold Matrix[2]</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>SIC Forward Transform Coeff Threshold Matrix[1]</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>SIC Forward Transform Coeff Threshold Matrix[0]</td>
</tr>
<tr>
<td>Values of the threshold matrix[0..6] are provided here.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold Matrix for 4x4 transform is as follows:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 4 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>31:24</td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>FBR SubPredMode Input</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>VME will use this to select the appropriate shapes from the input message to perform FME on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [1:0]: SubMbPredMode[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [3:2]: SubMbPredMode[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [5:4]: SubMbPredMode[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [7:6]: SubMbPredMode[3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00: Forward</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01: Backward</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10: Bidirectional</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11: Illegal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = FBR</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>FBR SubMBShape Input</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is used to specify the subshape per block for fractional and bidirectional refinement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [1:0]: SubMbShape[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [3:2]: SubMbShape[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [5:4]: SubMbShape[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [7:6]: SubMbShape[3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00: 8x8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01: 8x4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10: 4x8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11: 4x4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = FBR</td>
</tr>
<tr>
<td></td>
<td>7:2</td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>1:0</td>
<td></td>
<td><strong>FBR MbMode Input</strong>&lt;br&gt;This field is used to specify the inter macroblock type in the same format as VME output.&lt;br&gt;00: 16x16&lt;br&gt;01: 16x8&lt;br&gt;10: 8x16&lt;br&gt;11: 8x8&lt;br&gt;ValidMsgType = FBR</td>
</tr>
<tr>
<td>M2.4</td>
<td>31:24</td>
<td><strong>MV 7 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>MV 6 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>MV 5 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>MV 4 Cost</strong></td>
</tr>
<tr>
<td>M2.3</td>
<td>31:24</td>
<td><strong>MV 3 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>MV 2 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>MV 1 Cost</strong></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>MV 0 Cost</strong>&lt;br&gt;Motion vector costings. See 6.3.3.1 for details. In short, the cost is linearly interpolated between control points.&lt;br&gt;Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td>M2.2</td>
<td>31:24</td>
<td><strong>Chroma Intra Mode Cost</strong>&lt;br&gt;Penalty for chroma intra modes.&lt;br&gt;DC = 0x&lt;br&gt;Horz = 1x&lt;br&gt;Vert = 1x&lt;br&gt;Plane = 2x&lt;br&gt;Format = U4U4 (encoded value must fit in 12-bits)&lt;br&gt;ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>RefID Cost</strong>&lt;br&gt;RefID costing base penalty. Under AVC or Linear mode, different scaling are applied on top of this.&lt;br&gt;Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>Mode 9 Cost</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_BWD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>Mode 8 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_16x16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>M2.1</td>
<td>31:24</td>
<td><strong>Mode 7 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_4x4q</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_FIELD_8x8q</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td><strong>Mode 6 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_8x4q</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_4x8q</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_FIELD_16x8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td><strong>Mode 5 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_8x8q</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>Mode 4 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_16x8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTER_8x16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>M2.0</td>
<td>31:24</td>
<td><strong>Mode 3 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTRA_4x4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ValidMsgType = SIC, IME, FBR</strong></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>Mode 2 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTRA_8x8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>Mode 1 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTRA_16x16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 12-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Mode 0 Cost</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE_INTRA_NONPRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4U4 (encoded value must fit in 10-bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td>M3.7</td>
<td>31:0</td>
<td><strong>BWD Cost Center 3</strong></td>
</tr>
<tr>
<td>M3.6</td>
<td>31:0</td>
<td><strong>FWD Cost Center 3</strong></td>
</tr>
<tr>
<td>M3.5</td>
<td>31:0</td>
<td><strong>BWD Cost Center 2</strong></td>
</tr>
<tr>
<td>M3.4</td>
<td>31:0</td>
<td><strong>FWD Cost Center 2</strong></td>
</tr>
<tr>
<td>M3.3</td>
<td>31:0</td>
<td><strong>BWD Cost Center 1</strong></td>
</tr>
<tr>
<td>M3.2</td>
<td>31:0</td>
<td><strong>FWD Cost Center 1</strong></td>
</tr>
<tr>
<td>M3.1</td>
<td>31:16</td>
<td><strong>BWD Cost Center 0 Delta Y (BWDCostCenter0Y)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field defines the Y value for the first cost center relative to the picture source MB Y value for the BWD direction.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>BWD Cost Center 0 Delta X (BWDCostCenter0X)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field defines the X value for the first cost center relative to the picture source MB X value for the BWD direction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Major shape mapping to each cost center:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CC0: 16x16_0, 16x8_0, 8x16_0, 8x8_0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CC1: 8x16_1, 8x8_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CC2: 16x8_1, 8x8_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CC3: 8x8_3</td>
</tr>
</tbody>
</table>
### 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 3.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+0.6</td>
<td>31:0</td>
<td>Ref0 SkipCenter 3 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.5</td>
<td>31:0</td>
<td>Ref1 SkipCenter 2 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.4</td>
<td>31:0</td>
<td>Ref0 SkipCenter 2 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.3</td>
<td>31:0</td>
<td>Ref1 SkipCenter 1 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.2</td>
<td>31:0</td>
<td>Ref0 SkipCenter 1 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.1</td>
<td>31:0</td>
<td>Ref1 SkipCenter 0 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+0.0</td>
<td>31:0</td>
<td>Ref0 SkipCenter 0 Delta XY <em>(for definition see M3.7)</em></td>
</tr>
<tr>
<td>WX+1.7</td>
<td>31:0</td>
<td>Neighbor pixel Luma value [23, -1] to [20, -1]. Upper-right pixels from neighbor macroblock C</td>
</tr>
<tr>
<td>WX+1.6</td>
<td>31:0</td>
<td>Neighbor pixel Luma value [19, -1] to [16, -1]. Upper-right edge pixels from neighbor macroblock C</td>
</tr>
</tbody>
</table>

For chroma skip:
Format = S12.3 (2's comp)
Hardware Range: [-256.000 to 255.875]

Ref1SkipCenter3 Delta X;
This field defines the X value for the forward skip center relative to the 8x8 block offset from the source MB X location in quarter-pel precision associated with Ref1.
To match the relative 8x8 block location, the HW will add fixed offsets to the 4 skip centers in each direction to generate the correct pixel location to fetch the data.
For SkipCenter 0: VME will add 0 to the user-input X value.
For SkipCenter 1: VME will add 32 to the user-input X value.
For SkipCenter 2: VME will add 0 to the user-input X value.
For SkipCenter 3: VME will add 32 to the user-input X value.

Format = S13.2 (2's comp)
Hardware Range: [-2048.00 to 2047.75]

For chroma skip:
Format = S12.3 (2's comp)
Hardware Range: [-1024.000 to 1023.875]
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+1.5</td>
<td>31:0</td>
<td><strong>Neighbor pixel Luma value [15, -1] to [12, -1]</strong>. Top edge pixels from neighbor macroblock B</td>
</tr>
<tr>
<td>WX+1.4</td>
<td>31:0</td>
<td><strong>Neighbor pixel Luma value [11, -1] to [8, -1]</strong>. Top edge pixels from neighbor macroblock B</td>
</tr>
<tr>
<td>WX+1.3</td>
<td>31:0</td>
<td><strong>Neighbor pixel Luma value [7, -1] to [4, -1]</strong>. Top edge pixels from neighbor macroblock B</td>
</tr>
<tr>
<td>WX+1.2</td>
<td>31:24</td>
<td><strong>Neighbor pixel Luma value [3, -1]</strong>. Fourth top edge pixel from neighbor macroblock B</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>Neighbor pixel Luma value [2, -1]</strong>. Third top edge pixel from neighbor macroblock B</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>Neighbor pixel Luma value [1, -1]</strong>. Second top edge pixel from neighbor macroblock B</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Neighbor pixel Luma value [0, -1]</strong>. First top edge pixel from neighbor macroblock B</td>
</tr>
<tr>
<td>WX+1.1</td>
<td>31:24</td>
<td><strong>Corner Neighbor pixel 0</strong>. Its content depends on IntraCornerSwap field. It swaps with Corner Neighbor pixel 1.</td>
</tr>
<tr>
<td></td>
<td>23:10</td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
</tbody>
</table>
| 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:
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+1.0</td>
<td>31:25</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>24:16</td>
<td><strong>AVC Intra 8x8 Mode Mask (Intra16x16ModeMask):</strong> Disables given intra mode as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx xxx1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx xx1x:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx x1xx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx 1xxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxx1 xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xx1x xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x x1xx xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x 1xxx xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 xxxx xxxx:</td>
</tr>
<tr>
<td></td>
<td>15:9</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>8:0</td>
<td><strong>AVC Intra 4x4 Mode Mask (Intra16x16ModeMask):</strong> Disables given intra mode as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx xxx1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx xx1x:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx x1xx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxxx 1xxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xxx1 xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x xx1x xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x x1xx xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x 1xxx xxxx:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 xxxx xxxx:</td>
</tr>
<tr>
<td>WX+2.7</td>
<td>31:24</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>Penalty for Intra4x4 non-DC prediction mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Penalty for Intra8x8 non-DC prediction mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Penalty for Intra16x16 non-DC prediction mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U8</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>WX+2.5</td>
<td>31:16</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>15:0</td>
<td><strong>Neighbor pixel Chroma value CbCr pair [-1, -1]</strong></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Corner neighbor pixel pair (CbCr pair, each U8).</td>
<td></td>
</tr>
<tr>
<td>WX+2.4</td>
<td>31:28</td>
<td><strong>Intra Predictor Mode for Neighbor B15 (IntraMxMPredModeB15):</strong> 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.</td>
</tr>
<tr>
<td>27:24</td>
<td><strong>Intra Predictor Mode for Neighbor B14 (IntraMxMPredModeB14):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>23:20</td>
<td><strong>Intra Predictor Mode for Neighbor B11 (IntraMxMPredModeB11):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>19:16</td>
<td><strong>Intra Predictor Mode for Neighbor B10 (IntraMxMPredModeB10):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td><strong>Intra Predictor Mode for Neighbor A15 (IntraMxMPredModeA15):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td><strong>Intra Predictor Mode for Neighbor A13 (IntraMxMPredModeA13):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td><strong>Intra Predictor Mode for Neighbor A7 (IntraMxMPredModeA7):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td><strong>Intra Predictor Mode for Neighbor A5 (IntraMxMPredModeA5):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intra Predictor Modes for Neighbor A and B are only used if <strong>MODE_INTRA_NOPRED</strong> is not zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For intra mode selection, bias is applied to the predicted mode if a predictor is present for a</td>
</tr>
</tbody>
</table>
partition. This is achieved by applying a penalty term MODE_INTRA_NONPRED defined in the VME state to the cost functions for non-predicted modes.

The predictor for a given partition is from its left neighbor and top neighbor. The intra decision for a partition serves as the predictor for the next partition in the partition order as defined in Numbers of Block4x4 in a 16x16 region and Numbers of Block4x4 in an 8x8 region or numbers of Block8x8 in a 16x16 region.

This set of intra predictor mode for neighbor macroblocks are only used for INTRA8x8 and INTRA4x4 modes.

Format : U4 (The value of this field is defined in Definition of Intra4x4PredMode which is the same as that in Definition of Intra8x8PredMode.)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+2.3</td>
<td>31:24</td>
<td>Corner Neighbor pixel 1. Its content depends on IntraCornerSwap field. It swaps with Corner Neighbor pixel 0. Neighbor pixel Luma value [-1, -1]. The one upper-left edge pixel from neighbor macroblock D, which is the right most edge pixel of D, if IntraCornerSwap field is 1. Or 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.</td>
</tr>
<tr>
<td>WX+2.2</td>
<td>31:0</td>
<td>Neighbor pixel Luma value [-1, 14] to [-1, 12]. Left edge pixels from neighbor macroblock A</td>
</tr>
<tr>
<td>WX+2.1</td>
<td>31:0</td>
<td>Neighbor pixel Luma value [-1, 11] to [-1, 8]. Left edge pixels from neighbor macroblock A</td>
</tr>
<tr>
<td>WX+2.0</td>
<td>31:24</td>
<td>Neighbor pixel Luma value [-1, 7] to [-1, 4]. Left edge pixels from neighbor macroblock A</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>Neighbor pixel Luma value [-1, 2]. Third left edge pixel from neighbor macroblock A</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Neighbor pixel Luma value [-1, 1]. Second left edge pixel from neighbor macroblock A</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Neighbor pixel Luma value [-1, 0]. First left edge pixel from neighbor macroblock A</td>
</tr>
<tr>
<td>WX+3.7</td>
<td>31:0</td>
<td>Neighbor pixel Chroma value CbCr pair [7, -1] to [6, -1]</td>
</tr>
<tr>
<td>WX+3.6</td>
<td>31:0</td>
<td>Neighbor pixel Chroma value CbCr pair [5, -1] to [4, -1]</td>
</tr>
<tr>
<td>WX+3.5</td>
<td>31:0</td>
<td>Neighbor pixel Chroma value CbCr pair [3, -1] to [2, -1]</td>
</tr>
<tr>
<td>WX+3.4</td>
<td>31:0</td>
<td>Neighbor pixel Chroma value CbCr pair [1, -1] to [0, -1]</td>
</tr>
<tr>
<td>WX+3.3</td>
<td>31:0</td>
<td>Neighbor pixel Chroma value CbCr pair [-1, 7] to [-1, 6]</td>
</tr>
</tbody>
</table>
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 3.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+0.7</td>
<td>31:0</td>
<td>IME Search Path Delta 28-31</td>
</tr>
<tr>
<td>WX+0.6</td>
<td>31:0</td>
<td>IME Search Path Delta 24-27</td>
</tr>
<tr>
<td>WX+0.5</td>
<td>31:0</td>
<td>IME Search Path Delta 20-23</td>
</tr>
<tr>
<td>WX+0.4</td>
<td>31:0</td>
<td>IME Search Path Delta 16-19</td>
</tr>
<tr>
<td>WX+0.3</td>
<td>31:0</td>
<td>IME Search Path Delta 12-15</td>
</tr>
<tr>
<td>WX+0.2</td>
<td>31:0</td>
<td>IME Search Path Delta 8-11</td>
</tr>
<tr>
<td>WX+0.1</td>
<td>31:0</td>
<td>IME Search Path Delta 4-7</td>
</tr>
<tr>
<td>WX+0.0</td>
<td>31:0</td>
<td>IME Search Path Delta 0-3</td>
</tr>
<tr>
<td>WX+1.7</td>
<td>31:0</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>WX+1.6</td>
<td>31:0</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td>WX+1.5</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 52-55</strong></td>
</tr>
<tr>
<td>WX+1.4</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 48-51</strong></td>
</tr>
<tr>
<td>WX+1.3</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 44-47</strong></td>
</tr>
<tr>
<td>WX+1.2</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 40-43</strong></td>
</tr>
<tr>
<td>WX+1.1</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 36-39</strong></td>
</tr>
<tr>
<td>WX+1.0</td>
<td>31:0</td>
<td><strong>IME Search Path Delta 32-35</strong></td>
</tr>
<tr>
<td>WX+2.7</td>
<td>31:0</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td>WX+2.6</td>
<td>31:28</td>
<td><strong>Rec0 Shape 8x8_3 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td><strong>Rec0 Shape 8x8_2 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td><strong>Rec0 Shape 8x8_1 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Rec0 Shape 8x8_0 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td><strong>Rec0 Shape 8x16_1 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td><strong>Rec0 Shape 8x16_0 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td><strong>Rec0 Shape 16x8_1 RefID</strong></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td><strong>Rec0 Shape 16x8_0 RefID</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td>WX+2.5</td>
<td>31:16</td>
<td><strong>Rec0 Shape 16x16 Y (relative to source MB)</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Rec0 Shape 16x16 X (relative to source MB)</strong></td>
</tr>
<tr>
<td>WX+2.4</td>
<td>31:20</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td><strong>Rec0 Shape 16x16 RefID</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Rec0 Shape 16x16 Distortion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+2.3</td>
<td>31:16</td>
<td><strong>Rec0 Shape 8x8_3 Distortion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 14 bits. Upper bits ignored (True for all 8x8_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Rec0 Shape 8x8_2 Distortion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+2.2</td>
<td>31:16</td>
<td><strong>Rec0 Shape 8x8_1 Distortion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x8_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+2.1</td>
<td>31:16</td>
<td>Rec0 Shape 8x16_1 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 15 bits. Upper bits ignored (True for all 8x16_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x16_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+2.0</td>
<td>31:16</td>
<td>Rec0 Shape 16x8_1 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 15 bits. Upper bits ignored (True for all 16x8_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 16x8_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+3.7</td>
<td>31:16</td>
<td>Rec0 Shape 8x8_3 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x8_3 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.6</td>
<td>31:16</td>
<td>Rec0 Shape 8x8_2 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x8_2 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.5</td>
<td>31:16</td>
<td>Rec0 Shape 8x8_1 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x8_1 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.4</td>
<td>31:16</td>
<td>Rec0 Shape 8x8_0 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x8_0 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.3</td>
<td>31:16</td>
<td>Rec0 Shape 8x16_1 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x16_1 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.2</td>
<td>31:16</td>
<td>Rec0 Shape 8x16_0 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 8x16_0 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.1</td>
<td>31:16</td>
<td>Rec0 Shape 16x8_1 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 16x8_1 (relative to source MB)</td>
</tr>
<tr>
<td>WX+3.0</td>
<td>31:16</td>
<td>Rec0 Shape 16x8_0 (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec0 Shape 16x8_0 (relative to source MB)</td>
</tr>
<tr>
<td>WX+4.7</td>
<td>31:0</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td>WX+4.6</td>
<td>31:28</td>
<td>Rec1 Shape 8x8_3 RefID</td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Rec1 Shape 8x8_2 RefID</td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Rec1 Shape 8x8_1 RefID</td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Rec1 Shape 8x8_0 RefID</td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Rec1 Shape 8x16_1 RefID</td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Rec1 Shape 8x16_0 RefID</td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Rec1 Shape 16x8_1 RefID</td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Rec1 Shape 16x8_0 RefID</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td>WX+4.5</td>
<td>31:16</td>
<td>Rec1 Shape 16x16 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 16x16 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+4.4</td>
<td>31:20</td>
<td>Reserved MBZ</td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Rec1 Shape 16x16 RefID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 16x16 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+4.3</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_3 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 14 bits. Upper bits ignored (True for all 8x8_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_2 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+4.2</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_1 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+4.1</td>
<td>31:16</td>
<td>Rec1 Shape 8x16_1 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 15 bits. Upper bits ignored (True for all 8x16_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x16_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+4.0</td>
<td>31:16</td>
<td>Rec1 Shape 16x8_1 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware only uses 15 bits. Upper bits ignored (True for all 16x8_X Distortions).</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 16x8_0 Distortion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td>WX+5.7</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_3 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_3 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+5.6</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_2 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_2 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+5.5</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_1 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_1 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+5.4</td>
<td>31:16</td>
<td>Rec1 Shape 8x8_0 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x8_0 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+5.3</td>
<td>31:16</td>
<td>Rec1 Shape 8x16_1 Y (relative to source MB)</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Rec1 Shape 8x16_1 X (relative to source MB)</td>
</tr>
<tr>
<td>WX+5.2</td>
<td>31:16</td>
<td>Rec1 Shape 8x16_0 Y (relative to source MB)</td>
</tr>
</tbody>
</table>

422
### 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 3.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+0.7</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 3</td>
</tr>
<tr>
<td>WX+0.6</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 3</td>
</tr>
<tr>
<td>WX+0.5</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 2</td>
</tr>
<tr>
<td>WX+0.4</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 2</td>
</tr>
<tr>
<td>WX+0.3</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 1</td>
</tr>
<tr>
<td>WX+0.2</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 1</td>
</tr>
<tr>
<td>WX+0.1</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 0</td>
</tr>
<tr>
<td>WX+0.0</td>
<td>31:1</td>
<td>Ref0 Sub-block Y 0</td>
</tr>
</tbody>
</table>

**Format = S13.2 (2's comp)**

**Hardware Range:** [-2048.00 to 2047.75]
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit(s)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Ref0 Sub-block X 0</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The x-coordinate of Motion Vector 0 for Reference 0, relative to source MB location.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> 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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = S13.2 (2's comp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware Range: [-2048.00 to 2047.75]</td>
</tr>
<tr>
<td>WX+1.7</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 7</strong></td>
</tr>
<tr>
<td>WX+1.6</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 7</strong></td>
</tr>
<tr>
<td>WX+1.5</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 6</strong></td>
</tr>
<tr>
<td>WX+1.4</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 6</strong></td>
</tr>
<tr>
<td>WX+1.3</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 5</strong></td>
</tr>
<tr>
<td>WX+1.2</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 5</strong></td>
</tr>
<tr>
<td>WX+1.1</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 4</strong></td>
</tr>
<tr>
<td>WX+1.0</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 4</strong></td>
</tr>
<tr>
<td>WX+2.7</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 11</strong></td>
</tr>
<tr>
<td>WX+2.6</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 11</strong></td>
</tr>
<tr>
<td>WX+2.5</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 10</strong></td>
</tr>
<tr>
<td>WX+2.4</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 10</strong></td>
</tr>
<tr>
<td>WX+2.3</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 9</strong></td>
</tr>
<tr>
<td>WX+2.2</td>
<td>31:0</td>
<td><strong>Ref0 Sub-block XY 9</strong></td>
</tr>
<tr>
<td>WX+2.1</td>
<td>31:0</td>
<td><strong>Ref1 Sub-block XY 8</strong></td>
</tr>
</tbody>
</table>

424
### Return Data Message Phases

**Major changes:**

- Many of the fields are not valid output for all message types.
- Addtion 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 that 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.*

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit( s)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WX+2.0</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 8</td>
</tr>
<tr>
<td>WX+3.7</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 15</td>
</tr>
<tr>
<td>WX+3.6</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 15</td>
</tr>
<tr>
<td>WX+3.5</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 14</td>
</tr>
<tr>
<td>WX+3.4</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 14</td>
</tr>
<tr>
<td>WX+3.3</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 13</td>
</tr>
<tr>
<td>WX+3.2</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 13</td>
</tr>
<tr>
<td>WX+3.1</td>
<td>31:0</td>
<td>Ref1 Sub-block XY 12</td>
</tr>
<tr>
<td>WX+3.0</td>
<td>31:0</td>
<td>Ref0 Sub-block XY 12</td>
</tr>
</tbody>
</table>

### Description

- **Total VME Stalled Clocks:** Counts the number of clocks VME is stalled/starved while processing this request, due to cache misses.
  - Format: U16
  - ValidMsgType = SIC, IME, FBR

- **Total VME Compute Clocks:** Counts the number of clocks VME is processing this request, but not stalled/starved as a result of cache misses.
  - Format: U16
  - ValidMsgType = SIC, IME, FBR

- **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
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>MaxMV Occurred:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is set if the MaxMV event prevented the lowest distortion solution is rejected due to lack of motion vectors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid only for Luma Source Size = 16x16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = IME</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>EarlyIMEStop Occurred:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is set if the EarlyIMEStop threshold is satisfied and IME discontinues searching.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = IME</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is derived from sub_mb_type for a BP_8x8 macroblock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is derived from MbType for a non-BP_8x8 inter macroblock, and carries redundant information as MbType).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [1:0]: SubMbPredMode[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [3:2]: SubMbPredMode[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [5:4]: SubMbPredMode[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits [7:6]: SubMbPredMode[3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC, IME, FBR</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is derived from sub_mb_type for a BP_8x8 or equivalent macroblock.</td>
</tr>
</tbody>
</table>
|       |       | This field is forced to 0 for a non-BP_8x8 inter macroblock, and effectively carries redundant information.
This field is only valid if InterMbMode is INTER8x8. Otherwise, it is set to zero.

Bits [1:0]: SubMbShape[0]
Bits [3:2]: SubMbShape[1]
Bits [5:4]: SubMbShape[2]
Bits [7:6]: SubMbShape[3]
ValidMsgType = SIC, IME, FBR

Macroblock Intra Structure (MbIntraStruct): This is a bitmask specifies neighbor macroblock availability. This allows software to constrain intra prediction mode search.

This field is simply copied from the input message (to reduce software overhead of forming the output message to PAK).

<table>
<thead>
<tr>
<th>Bits</th>
<th>MotionVerticalFieldSelect Index</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

Note: This 8b field is MBZ when IntraComputeType == 1X (when intra is disabled).
ValidMsgType = SIC

W0.5  31:16  LumaIntraPredModes[3]
      15:0  LumaIntraPredModes[2]
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
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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**  
The IntraMbMode will indicate 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 |
<p>| W0.1  | 31:27 | <strong>Reserved: MBZ</strong> |
| 26:16 | | <strong>Sum Ref1 Inter Dist Upper 10 bits (SumInterDistL1Upper)</strong> |
| 15:8  | | <strong>Search Path Length:</strong> This field returns the number of SU it takes in the integer search. It includes predetermined search path and dynamic search path. |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
|       |     | **Format:** U8  
|       |     | ValidMsgType = IME |
| 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  | **Reserved:** MBZ |
|       | 30  | **Reserved:** MBZ |
|       | 29  | **Reserved:** MBZ |
|       | 28:24 | **MvQuantity**  
|       |     | Specify the number of MVs in packed format (in unit of motion vectors).  
|       |     | *Note:* This field is provided to help with software to 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.**  
<p>|       |     | 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. |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22:21</td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
</tbody>
</table>
|       | 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 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 will be always 0 for non-16x16 source block cases.  
ValidMsgType = IME, FBR |
|       | 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 an unified encoding for inter macroblocks according to AVC Spec.  
ValidMsgType = SIC, IME, FBR |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 7     | 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`:  
\[ \text{SrcAccess} \land (\text{bit0}(\text{SrcY}) == 1). \]  
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     | 6   | **Reserved: MBZ** |
| 5:4   | 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   | 3:2 | **Reserved: MBZ** |
| 1:0   | 1:0 | **InterMbMode**  
This field is provided to carry redundant information as that in `MbType`. The full extended definition of this field allows kernel software to help update the `MbType` field when outputting controls to the MFX PAK encoding.  
VME outputs this field regardless of MbIntraFlag value if inter mode is enabled.  
ValidMsgType = SIC, IME, FBR |
| **W1.7 to W1.2** | **31:0 Each** | **MVb[3] to MVb[1]**. Motion vectors 3 to 1 for Reference 1, and  
**MVa[3] to MVa[1]**. Motion vectors 3 to 1 for Reference 0  
ValidMsgType = SIC, IME, FBR |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.  
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  
ValidMsgType = SIC, IME, FBR |
| W3.7 to W3.0 | 31:0 | Each  
**MVb[11] to MVb[8]**. Motion vectors 11 to 8 for Reference 1, and  
**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  
ValidMsgType = SIC, IME, FBR |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVa[11] to MVa[8]. Motion vectors 11 to 8 for Reference 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValidMsgType = SIC, IME, FBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W4.7 to W4.0</td>
<td>31:0 Each</td>
<td>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</td>
</tr>
<tr>
<td>ValidMsgType = SIC, IME, FBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W5.7 to W5.1</td>
<td>31:0 Each</td>
<td>InterDistortion[15] to InterDistortion[2]. Inter-prediction-distortion associated with motion vector 15 to 2. Its meaning is determined by sub-shape.</td>
</tr>
<tr>
<td>ValidMsgType = SIC, IME, FBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W5.0</td>
<td>31:16</td>
<td>InterDistortion[1]. Inter-prediction-distortion with motion vector 1 (co-located with subblock_4x4_1). Its meaning is determined by sub-shape.</td>
</tr>
<tr>
<td>Format = U16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValidMsgType = SIC, IME, FBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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 &quot;duplication&quot;, this field must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For 1MVP skip messages, the 16x16 distortion (sad + mv cost + ref cost) is present here.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For 4MVP skip messages, the 4 8x8 distortions (sad + mv cost + ref cost) are present here.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format = U16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W6.7</td>
<td>31:16</td>
<td>Max Ref1 Inter Dist (MaxRef1InterDist)</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Max Ref0 Inter Dist (MaxRef0InterDist)</td>
<td></td>
</tr>
<tr>
<td>W6.6</td>
<td>31:27</td>
<td>Reserved</td>
</tr>
<tr>
<td>26:0</td>
<td>Sum Ref0 Inter Dist (SumRef0InterDist)</td>
<td></td>
</tr>
<tr>
<td>W6.5</td>
<td>31:16</td>
<td>Block 0 Chroma Cr Coeff Magnitude Clip Sum</td>
</tr>
<tr>
<td>Sum of how much all the coefficients across 1 block exceeded their respective threshold.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: This field is MBZ when SkipModeEn is not set (when skip is disabled).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Block 0 Chroma Cb Coeff Magnitude Clip Sum</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sum of how much all the coefficients across 1 block exceeded their respective threshold.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This field is MBZ when SkipModeEn is not set (when skip is disabled).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC</td>
</tr>
<tr>
<td>W6.4</td>
<td>31:16</td>
<td><strong>Sum Ref1 Inter Dist lower 16 bits (SumInterDistL1Lower)</strong></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>Block 0 Chroma Cr NZC</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Count of the coefficients across 1 block that exceeded their respective threshold.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This field is MBZ when SkipModeEn is not set (when skip is disabled).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC</td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>Block 0 Chroma Cb NZC</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Count of the coefficients across 1 block that exceeded their respective threshold.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This field is MBZ when SkipModeEn is not set (when skip is disabled).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC</td>
</tr>
<tr>
<td>W6.3</td>
<td>31:16</td>
<td><strong>Block 3 Luma Coeff Magnitude Clip Sum</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Block 2 Luma Coeff Magnitude Clip Sum</strong></td>
</tr>
<tr>
<td>W6.2</td>
<td>31:16</td>
<td><strong>Block 1 Luma Coeff Magnitude Clip Sum</strong></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Block 0 Luma Coeff Magnitude Clip Sum</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sum of how much all the coefficients across 1 block exceeded their respective threshold.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This field is MBZ when SkipModeEn is not set (when skip is disabled).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidMsgType = SIC</td>
</tr>
<tr>
<td>W6.1</td>
<td>31:24</td>
<td><strong>Block 3 Luma NZC</strong></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td><strong>Block 2 Luma NZC</strong></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td><strong>Block 1 Luma NZC</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 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**

Note: IME Streamout follows the same format as the IME Streamin message phases (IME2-IME5).
### 3D Pipeline Stages

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Pipeline Stage</th>
<th>Functions Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Stream (CS)</td>
<td></td>
<td>The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads &quot;constant data&quot; from memory buffers and places it in the URB. Note that the CS stage is shared between the 3D, GPGPU and Media pipelines.</td>
</tr>
<tr>
<td>Vertex Fetch (VF)</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Vertex Shader (VS)</td>
<td></td>
<td>The Vertex Shader stage is responsible for processing (shading) incoming vertices by passing them to VS threads.</td>
</tr>
<tr>
<td>Hull Shader (HS)</td>
<td></td>
<td>The Hull Shader is responsible for processing (shading) incoming patch primitives as part of the tessellation process.</td>
</tr>
<tr>
<td>Tessellation Engine (TE)</td>
<td></td>
<td>The Tessellation Engine is responsible for using tessellation factors (computed in the HS stage) to tessellate U,V parametric domains into domain point topologies.</td>
</tr>
<tr>
<td>Domain Shader (DS)</td>
<td></td>
<td>The Domain Shader stage is responsible for processing (shading) the domain points (generated by the TE stage) into corresponding vertices.</td>
</tr>
<tr>
<td>Geometry Shader (GS)</td>
<td></td>
<td>The Geometry Shader stage is responsible for processing incoming objects by passing each object’s vertices to a GS thread.</td>
</tr>
<tr>
<td>Stream Output Logic (SOL)</td>
<td></td>
<td>The Stream Output Logic is responsible for outputting incoming object vertices into Stream Out Buffers in memory.</td>
</tr>
<tr>
<td>Clipper (CLIP)</td>
<td></td>
<td>The Clipper stage performs Clip Tests on incoming objects and clips objects if required. Objects are clipped using fixed-function hardware.</td>
</tr>
<tr>
<td>Strip/Fan (SF)</td>
<td></td>
<td>The Strip/Fan stage performs object setup. Object setup uses fixed-function hardware.</td>
</tr>
<tr>
<td>Windower/Masker (WM)</td>
<td></td>
<td>The Windower/Masker performs object rasterization and determines visibility coverage.</td>
</tr>
</tbody>
</table>
### 3D Pipeline-Level State

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

**Push Constant URB Allocation**

The push constants are stored into 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 bottom address of the URB and must ensure that none of the shaders have overlapping handles. Below is a diagram that represents a possible programming of the URB with Push Constants.

<table>
<thead>
<tr>
<th>Project</th>
<th>URB Allocation Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>The sizes of the regions in the diagram double to 32KB and 480KB, respectively.</td>
</tr>
</tbody>
</table>

**URB Allocation**

In the above scheme we are allocating 16KB of push constants and 240KB of URB space. The handle allocation is shown in the order of the FF pipeline but with the current hardware and state, the software can program these to be any order and may size them to zero. Software may also use some if not all of the 16KB above as handle allocations as long as none of the push constants or handle allocations overlap.
overlap. The only limitations are the sizes based off the table below and the restrictions in granularity which are specified in the command descriptions of the URB state and the push constant allocation state for each fixed function.

The next table specifies the maximum size of each buffer.

<table>
<thead>
<tr>
<th>Project</th>
<th>Max Constant Buffer</th>
<th>URB Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>32KB</td>
<td>512KB</td>
</tr>
<tr>
<td>HSW</td>
<td>16KB</td>
<td>128KB</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>API-level Statistic</th>
<th>HW Support</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IAVertices</strong> = # of vertices IA generated. May or may not include (a) vertices in partial primitives, (b) unused adjacent-only vertices. Not affected by vertex caching.</td>
<td>VF maintains <strong>IA_Vertices_COUNT</strong>. Will include unused adjacent-only vertices. Will not include vertices in partial primitives.</td>
</tr>
<tr>
<td><strong>IAPrimitives</strong> = # of primitives (objects) IA generated. May or may not include partial primitives.</td>
<td>VF maintains <strong>IA_Primitives_COUNT</strong>. 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).</td>
</tr>
<tr>
<td><strong>VSInvocations</strong> = # 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.</td>
<td>VS maintains <strong>VS_Invocation_COUNT</strong>. Impacted by vertex caching. Will not include vertices in partial primitives. Will 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.</td>
</tr>
<tr>
<td><strong>HSInvocations</strong> = # of patches executed by HS.</td>
<td>HS maintains <strong>HS_Invocation_COUNT</strong>. This gets incremented by 1 for each patch whenever HS is enabled.</td>
</tr>
<tr>
<td><strong>DSInvocations</strong> = # 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.</td>
<td>DS maintains <strong>DS_Invocation_COUNT</strong>. This is incremented for each domain point passed to a DS thread.</td>
</tr>
<tr>
<td><strong>GSInvocations</strong> = # of times GS is executed. Obviously does not include partial primitives. May be incremented when StreamOut enabled, even if NULL_GS.</td>
<td>GS maintains <strong>GS_Invocation_COUNT</strong>, incrementing it by <strong>GSInvocations Increment Value</strong> for each dispatched instance. Will not be incremented if NULL_GS.</td>
</tr>
<tr>
<td><strong>GSPrimitives</strong> = # of primitives GS generated. Does not include primitives passing through a disabled GS stage. May or may</td>
<td>GS maintains <strong>GS_Primitive_COUNT</strong>. GS unit will increment this as it parses the GS thread output.</td>
</tr>
<tr>
<td>API-level Statistic</td>
<td>HW Support</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>not include partial primitives output by GS.</td>
<td>Will not include partial primitives output by GS threads.</td>
</tr>
<tr>
<td><strong>NumPrimitivesWritten[&lt;stream#&gt;]</strong> = # of complete primitives written to the stream's SO buffer, subject to buffer overflow.</td>
<td>SOL maintains <strong>SO_NUM_PRIMS_WRITTEN[0-3]</strong>.</td>
</tr>
<tr>
<td><strong>PrimitiveStorageNeeded[&lt;stream#&gt;]</strong> = # of complete primitives which would have been written to the stream's SO buffer ignoring any overflow.</td>
<td>SOL maintains <strong>SO_PRIM_STORAGE_NEEDED[0-3]</strong>.</td>
</tr>
<tr>
<td><strong>CInvocations</strong> = # of primitives entering 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.</td>
<td>CL OSB maintains <strong>CL_INVOCATION_COUNT</strong>.</td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td><strong>CPrimitives</strong> = # 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. Accomodates infinite or no guardband.</td>
<td>SF OSB maintains <strong>CL_PRIMITIVES_COUNT</strong>.</td>
</tr>
<tr>
<td>Will not include partial primitives.</td>
<td></td>
</tr>
<tr>
<td><strong>PSInvocations</strong> = # of times PS is executed, including unlit &quot;helper pixels&quot; within a subspan that need to go through the PS shader to provide 2x2 gradients. Accomodates early depth/stencil. Does not increment if NULL PS. Multisampling: counts pixels shaded If PERPIXEL or samples shaded if PERSAMPLE.</td>
<td>WIZ maintains <strong>PS_INVOCATION_COUNT</strong>.</td>
</tr>
<tr>
<td><strong>Occlusion</strong> = # of &quot;visible&quot; 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.</td>
<td>WIZ &amp; PBE maintain <strong>PS_DEPTH_COUNT</strong>.</td>
</tr>
</tbody>
</table>
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.

**Terminology Note:** 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*?

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.

**Notes:**

- 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

<table>
<thead>
<tr>
<th>3D Primitive Topology Type (ordered alphabetically)</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUADLIST</td>
<td>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. <strong>Programming Restrictions:</strong> Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUADSTRI P</td>
<td>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</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D Primitive Topology Type (ordered alphabetically)</td>
<td>Description</td>
<td>Project Security</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td><strong>3D Primitive Topology Type</strong> ordered alphabetically</td>
<td><strong>Description</strong></td>
<td><strong>Project Security</strong></td>
<td></td>
</tr>
<tr>
<td><strong>object.</strong></td>
<td><strong>Programming Restrictions:</strong> Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RECTLIST</strong></td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Programmng Restrictions:</strong> Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The RECTLIST primitive is supported specifically for 2D operations (e.g., BLTs and &quot;stretch&quot; BLTs) and not as a general 3D primitive. Due to this, a number of restrictions apply to the use of RECTLIST:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Must utilize &quot;screen space&quot; coordinates (VPOS_SCREENSAPCE) 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).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Viewport Mapping must be DISABLED (as is typical with the use of screen-space coordinates).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TRIFAN</strong></td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Programmng Restrictions:</strong> Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TRIFAN_NO STIPPLE</strong></td>
<td>Similar to TRIFAN, but polygon stipple is not applied (even if enabled). This can be used to support &quot;point&quot; polygon fill mode, under the combination of the following conditions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D Primitive Topology Type (ordered alphabetically)</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(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 &quot;point&quot; and the other is &quot;solid&quot;, (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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRILIST</td>
<td>A list of independent triangle objects (3 vertices per triangle). <strong>Programming Restrictions:</strong> Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRILIST_ADJ</td>
<td>A list of independent triangle objects with adjacency information (6 vertices per triangle). <strong>Programming Restrictions:</strong> Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. Not valid as output from GS thread.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRISTRIP</td>
<td>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. <strong>Programming Restrictions:</strong> Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_ADJ</td>
<td>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. <strong>Programming Restrictions:</strong> Normal usage expects at least 6 vertices, though incomplete objects are silently ignored. Not valid as output from GS thread.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D Primitive Topology Type (ordered alphabetically)</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>TRISTRIP_REVERSE</td>
<td>Similar to TRISTRIP, though the sense of orientation (winding order) is reversed – this allows SW to break long tristrips into smaller pieces and still maintain correct face orientations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PATCHLIST_n</td>
<td>List of n-vertex &quot;patch&quot; 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.)</td>
<td>HSW</td>
<td></td>
</tr>
</tbody>
</table>

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).
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).
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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>31:0</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>
| D1    | 31:0 | **Render Target Array Index (RTAIndex).** This value is (eventually) used to index into a specific element of an array Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function.

Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required, software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a range check of computed index values against \([0,2047]\), and output zero if that range is exceeded. Note that the unmodified renderTargetArrayIndex must be maintained in the VUE outside of the Vertex Header.

Software can force an RTAIndex of 0 to be used (effectively ignoring the setting of this DWord) by use of the **ForceZeroRTAIndex** bit (3DSTATE_CLIP). Otherwise the read-back value will be used to select an RTArray element, after being clamped to the RTArray surface’s \([\text{MinimumArrayElement}, \text{Depth}]\) range (SURFACE_STATE).

Format: 0-based U32 index value |
| D2    | 31:0 | **Viewport Index.** This value is used to select one of a possible 16 sets of viewport (VP) state parameters in the Clip unit’s VertexClipTest function and in the SF unit’s ViewportMapping and Scissor functions.

The GS unit (even if disabled) will read back this value for all vertices exiting the GS stage and entering the Clip stage. When enabled, the GS unit will range-check the value against \([0, \text{Maximum VPIndex}]\) (see GS_STATE, CLIP_STATE). After this range-check the values are sent down the pipeline and used in the Clip unit’s VertexClipTest function. For vertices passing through the Clip stage, these values will also be sent to the SF unit for use in ViewportMapping and Scissor functions.

The Clip unit (if enabled) will read back this value only for vertices generated by CLIP threads. The Clip unit will perform a range clamp similar to the GS unit.

Software can force a value of 0 to be used by programming **Maximum VPIndex** to 0.

Format: 0-based U32 index value |
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>31:0</td>
<td><strong>Point Width.</strong> This field specifies the width of POINT objects in screen-space pixels. It is used only for vertices within POINTLIST and POINTLIST_BF primitive topologies, and is ignored for vertices associated with other primitive topologies. This field is read back by both the GS and Clip units. Format: FLOAT32</td>
</tr>
<tr>
<td>D4</td>
<td>31:0</td>
<td><strong>Vertex Position X Coordinate.</strong> This field contains the X component of the vertex's 4D space position. Format: FLOAT32</td>
</tr>
<tr>
<td>D5</td>
<td>31:0</td>
<td><strong>Vertex Position Y Coordinate.</strong> This field contains the Y component of the vertex's 4D space position. Format: FLOAT32</td>
</tr>
<tr>
<td>D6</td>
<td>31:0</td>
<td><strong>Vertex Position Z Coordinate.</strong> This field contains the Z component of the vertex's NDC space position. Format: FLOAT32</td>
</tr>
<tr>
<td>D7</td>
<td>31:0</td>
<td><strong>Vertex Position W Coordinate.</strong> This field contains the Z component of the vertex's 4D space position. Format: FLOAT32</td>
</tr>
<tr>
<td>D8</td>
<td>31:0</td>
<td><strong>ClipDistance 0 Value (optional).</strong> 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&lt;0&gt; bit will set in the Clip unit’s VertexClipTest function. If the UserClipDistance Clip Test Enable Bitmask bit is clear, this value will not be read back, and the vertex’s UCF&lt;0&gt; bit will be zero by definition. Format: FLOAT32</td>
</tr>
<tr>
<td>D9</td>
<td>31:0</td>
<td><strong>ClipDistance 1 Value (optional).</strong> See above</td>
</tr>
<tr>
<td>D10</td>
<td>31:0</td>
<td><strong>ClipDistance 2 Value (optional).</strong> See above</td>
</tr>
<tr>
<td>D11</td>
<td>31:0</td>
<td><strong>ClipDistance 3 Value (optional).</strong> See above</td>
</tr>
<tr>
<td>D12</td>
<td>31:0</td>
<td><strong>ClipDistance 4 Value (optional).</strong> See above</td>
</tr>
<tr>
<td>D13</td>
<td>31:0</td>
<td><strong>ClipDistance 5 Value (optional).</strong> See above</td>
</tr>
<tr>
<td>D14</td>
<td>31:0</td>
<td><strong>ClipDistance 6 Value (optional).</strong> See above</td>
</tr>
</tbody>
</table>
### 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_CLIPSSPACE (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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D15</td>
<td>31:0</td>
<td><strong>ClipDistance 7 Value (optional).</strong> See above</td>
</tr>
<tr>
<td></td>
<td>31:0</td>
<td><strong>(Remainder of Vertex Elements).</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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) (<strong>Vertex Entry URB Read Length</strong> 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.</td>
</tr>
</tbody>
</table>
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).
3D Pipeline – Vertex Fetch (VF) Stage

Vertex Fetch (VF) Stage Overview

The VF stage performs one major function: executing 3DPRIMITIVE commands. This is handled by the VF’s InputAssembly function.

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

State

This section contains various state registers.

Control State

3DSTATE_VF

Index Buffer (IB) State

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

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

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software is responsible for ensuring that accesses outside the IB do not occur. This is possible as software can compute the range of IB values referenced by a 3DPRIMITIVE command (knowing the StartVertexLocation, InstanceCount, and VerticesPerInstance values) and can then compare this range to the IB extent.</td>
<td></td>
</tr>
</tbody>
</table>

3DSTATE_INDEX_BUFFER

3DSTATE_INDEX_BUFFER

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>When 3DSTATE_VF has Cut Index Enable set, it is UNDEFINED to issue a 3DPRIMITIVE with a primitive topology type not supporting a Cut Index (even if no cut indices are actually present in the index buffer).</td>
</tr>
<tr>
<td>Definition</td>
<td>Cut Index?</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>3DPRIM_POINTLIST</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINELIST</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRILIST</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRISTRIP</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRIFAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>3DPRIM_QUADLIST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>3DPRIM_QUADSTRIP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>3DPRIM_LINELIST_ADJ</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP_ADJ</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRILIST_ADJ</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRISTRIP_ADJ</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRISTRIP.Reverse</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_POLYGON</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>3DPRIM_RECTLIST</td>
<td>N</td>
</tr>
<tr>
<td>3DPRIM_LINELOOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>3DPRIM_POINTLIST.BF</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP.CONT</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP.BF</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP.CONT.BF</td>
<td>Y</td>
</tr>
<tr>
<td>3DPRIM_TRIFAN.NOSTIPPLE</td>
<td>N</td>
</tr>
<tr>
<td>3DPRIM_PATCHLIST.n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>HSW</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Instead of &quot;VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch&quot;, the address of the byte immediately beyond the last valid byte of the buffer is determined by &quot;VBState.EndAddress + 1&quot;.</td>
</tr>
</tbody>
</table>
### VERTEXDATA Buffers – RANDOM Access

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Instead of &quot;VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch&quot;, the address of the byte immediately beyond the last valid byte of the buffer is determined by &quot;VBState.EndAddress + 1&quot;.</td>
</tr>
</tbody>
</table>

![Diagram](image)

### INSTANCEDATA Buffers

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Instead of &quot;VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch&quot;, the address of the byte immediately beyond the last valid byte of the buffer is determined by &quot;VBState.EndAddress + 1&quot;.</td>
</tr>
</tbody>
</table>
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 3DPRIMITIVIE processing in the VF unit.

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

**VERTEX_ELEMENT_STATE**

3DSTATE_VERTEX_ELEMENTS

3D_VertexComponentControl

Vertex Fetch State Preprocessing Extension

After creation of this topic is approved, will move the content for Vertex Fetch State Preprocessing Extension into it.
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_PrimTopoType

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 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.)

**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.

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 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.

### 3D Primitive Processing

#### Functional Overview

The following pseudocode summarizes the general flow of 3D Primitive Processing.

```plaintext
CommandInit
  InstanceLoop {
    VertexLoop {
      VertexIndexGeneration
      if ( cutFlag )
        TerminatePrimitive
      else {
        OutputBufferedVertex
        VertexCacheLookup
        if ( miss ) {
          VertexElementLoop {
            SourceElementFetch
            FormatConversion
            DestinationComponentSelection
            PrimitiveInfoGeneration
            URBWrite
          }
        }
      }
    }
  }
  TerminatePrimitive
```
**InstanceLoop**

The InstanceLoop is the outmost loop, iterating through each instance of primitives. There is no special "non-instanced" mode – at a minimum there is one instance of primitives.

For SEQUENTIAL accessing, the VertexID value is initialized to 0 at the start of each instance. (For RANDOM accessing, there is no initial value for VertexID, as it is derived from the fetched IB value).

The PrimitiveID is also initialized to 0 at the start of each instance. StartPrim is initialized to TRUE.

The VertexLoop (see below) is then executed to iterate through the instance vertices and output vertices to the pipeline as required.

The end of each iteration of InstanceLoop includes an implied "cut" operation.

The InstanceID value is incremented at the end of each InstanceLoop. Note that each instance will produce the same vertex outputs with the exception of any data dependent on InstanceID (i.e., "instance data").

**VertexLoop**

The VertexLoop iterates VertexNumber through the VertexCountPerInstance vertices for the instance.

For each iteration, a number of processing steps are performed (see below) to generate the information that comprises a vertex. Note that, due to CutProcessing, each iteration does not necessarily output a vertex to the pipeline. When a vertex is to be output, the following information is generated for that vertex:

- PrimitiveType associated with the vertex. This is simply a copy of the PrimitiveTopologyType field of the 3DPRIMITIVE
- VUE handle at which the vertex data is stored:
  - For a Vertex Cache hit, the VUE handle is marked with a VCHit boolean, so that the VS unit will not attempt to process (shade) that vertex.
  - Otherwise, the VertexLoop will generate and store the input vertex data into the VUE referenced by this handle.
- The PrimitiveID associated with the vertex. See PrimitiveInfoGeneration.
- PrimStart and PrimEnd booleans associated with the vertex. See PrimitiveInfoGeneration.

(Note that a single vertex of buffering is required in order to associate PrimEnd with a vertex, as this information may not be known until the next iteration through the VertexLoop (see OutputPrimitiveDelimiter).

VertexNumber value is incremented by 1 at the end of the loop.

**VertexIndexGeneration**

A VertexIndex value needs to be derived for each vertex. With the exception of the "cut" index, this index value is used as the vertex cache tag and as a structure index into all VERTEXDATA VBs.
For SEQUENTIAL accessing, the VertexID and VertexIndex value is derived as shown below:

\[
\text{VertexIndex} = \text{StartVertexLocation} + \text{VertexNumber} \\
\text{VertexID} = \text{VertexNumber}
\]

For RANDOM access, the VertexID and VertexIndex is derived from an IBValue read from the IB, as shown below:

\[
\begin{align*}
\text{IBIndex} &= \text{StartVertexLocation} + \text{VertexNumber} \\
\text{VertexID} &= \text{IB}[\text{IBIndex}] \\
\text{if ( CutIndexEnable && VertexID == CutIndex } \) \\
&\quad \text{CutFlag} = 1 \\
\text{else} \\
&\quad \text{VertexIndex} = \text{VertexID} + \text{BaseVertexLocation} \\
&\quad \text{CutFlag} = 0 \\
\end{align*}
\]

**Index Buffer Access**

The following figure illustrates how the Index Buffer is accessed.

**TerminatePrimitive**

For RANDOM accessing, and when enabled via Cut Index Enable, a fetched IBValue of 'all ones' (0xFF, 0xFFFF, or 0xFFFFFFFF depending on Index Format) is interpreted as a 'cut value' and signals the termination of the current primitive and the possible start of the next primitive. This allows the
application to specify an instance as a sequence of variable-sized strip primitives (though the cut value applies to any primitive type).

Also, there is an implied primitive termination at the end of each InstanceLoop (and so strip primitives cannot span multiple instances).

In either case, the currently-buffered vertex (if any) is marked with EndPrim and then flushed out to the pipeline.

The next-output vertex (if any) is marked with StartPrim.

Whenever a primitive delimiter is encountered, the PIDCounterS and PIDCounterR counters are reset to 0. These counters control the incrementing (in PrimitiveInfoGeneration, below) of PrimitiveID within each primitive topology of an instance.

```c
if ( PIDCounterS != 0 ) // There is a buffered vertex
    if ( primType == TRISTRIP_ADJ )
        if ( PIDCounterS== 6 || PIDCounterR == 1 )
            PrimitiveID ++
        endif
    endif
    PrimEnd = TRUE
    OutputBufferedVertex
endif
PrimEnd = FALSE
PrimStart = TRUE
```

**VertexCacheLookup**

The VertexIndex value is used as the tag value for the VertexCache (see *Vertex Cache* above). If the Vertex Cache is enabled and the VertexIndex value hits in the cache, the VUE handle is read from the cache and inserted into the vertex stream. It is marked with a VCHit boolean to suppress processing (shading) in the VS unit.

Otherwise, for Vertex Cache misses, a VUE handle is obtained to provide storage for the generated vertex data. VertexLoop processing then proceeds to iterate through the VEs to generate the destination VUE data.

**VertexElementLoop**

The VertexElementLoop generates and stores vertex data in the destination VUE one VE at a time.

**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.
**SourceElementFetch**

The following assumes the VE requires data from a VB, which is the typical case. In the case that the VE is completely comprised of constant and/or auto-generated IDs, the SourceElementFetch and FormatConversion steps are skipped.

The structure index within the VE’s selected VB is computed as follows:

```plaintext
if (VB is a VERTEXDATA VB)
    VBIndex = VertexIndex
else // INSTANCEDATA VB
    VBIndex = StartInstanceLocation
    if (VB.InstanceDataStepRate > 0)
        VBIndex += InstanceID/VB.InstanceDataStepRate
    endif
endif
```

If VBIndex is invalid (i.e., negative or past Max Index), the data returned from the VB fetch is defined to be zero. Otherwise, the address of the source data required for the VE is then computed and the data is read from the VB. The amount of data read from the VB is determined by the **Source Element Format**.
if ( (VBIndex < 0) || (VBIndex > VB.MaxIndex) )
    srcData = 0
else
    pSrcData = VB.BufferStartingAddress + (VBIndex * VB.BufferPitch) + VE.SourceElementOffset
    srcData = MemoryRead(pSrcData, VE.SourceElementFormat)
endif

Format Conversion

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Source Element</th>
<th>Converted Component</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Surface Format Name</td>
<td>Format</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R32G32B32A32_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R64G64_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R32G32B32A32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32B32A32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td>Project</td>
<td>Source Element</td>
<td>Converted Component</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>Surface Format Name</td>
<td>Format</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R32G32_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R32G32_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R64_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16A16_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32G32_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>HSW:8+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R32G32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>B8G8R8A8_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R10G10B10A2_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R10G10B10A2_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R10G10B10_SNORM_A2_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R16G16_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R16G16_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>HSW:8+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B10G10R10A2_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R11G11B10_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td></td>
<td>R32_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R32_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R32_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R10G10B10X2_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R8G8B8A8_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>Project</td>
<td>Source Element</td>
<td>Converted Component</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>Surface Format Name</td>
<td>Format</td>
</tr>
<tr>
<td>R16G16_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R16G16_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R32_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R32_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_UNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_SNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_SINT</td>
<td>SINT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_UINT</td>
<td>UINT</td>
<td>R</td>
</tr>
<tr>
<td>R16_UNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R16_SNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R16_SINT</td>
<td>SINT</td>
<td>R</td>
</tr>
<tr>
<td>R16_UINT</td>
<td>UINT</td>
<td>R</td>
</tr>
<tr>
<td>R16_FLOAT</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R16_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R16_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8_UNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8_SNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8_SINT</td>
<td>SINT</td>
<td>R</td>
</tr>
<tr>
<td>R8_UINT</td>
<td>UINT</td>
<td>R</td>
</tr>
<tr>
<td>R8_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8B8_UNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8B8_SNORM</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8B8_SSCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>R8G8B8_USCALED</td>
<td>FLOAT</td>
<td>R</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R8G8B8_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R8G8B8_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td></td>
<td>R64G64B64A64_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R64G64B64_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_UNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>R16G16B16_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R16G16B16_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td>HSW:B+</td>
<td>R16G16B16_SINT</td>
<td>SINT</td>
</tr>
</tbody>
</table>
### Project

<table>
<thead>
<tr>
<th>Source Element</th>
<th>Converted Component</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Surface Format Name</strong></td>
<td><strong>Format</strong></td>
</tr>
<tr>
<td>R32_SFIXED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>R10G10B10A2_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td>R10G10B10A2_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>R10G10B10A2_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>R10G10B10A2_SINT</td>
<td>SINT</td>
</tr>
<tr>
<td>B10G10R10A2_SNORM</td>
<td>FLOAT</td>
</tr>
<tr>
<td>B10G10R10A2_USCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>B10G10R10A2_SSCALED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>B10G10R10A2_UINT</td>
<td>UINT</td>
</tr>
<tr>
<td>B10G10R10A2_SINT</td>
<td>SINT</td>
</tr>
</tbody>
</table>

### DestinationFormatSelection

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

### PrimitiveInfoGeneration

A **PrimitiveID value** and PrimStart boolean need to be associated with the vertex.

If the vertex is either the first vertex of an instance or the first vertex following a 'cut index', the vertex is marked with PrimStart.

PrimitiveID gets incremented such that subsequent per-object processing (i.e., in the GS or SF/WM) sees an incrementing value associated with each sequential object within an instance. The PrimitiveID associated with the provoking, non-adjacent vertex of an object is applied to the object.

The following pseudocode describe the logic used in the VertexLoop to compute the PrimitiveID value associated with the vertex. Recall that PrimitiveID is reset to 0 at the start of each InstanceLoop.

```plaintext
if ( PIDCounterS < S[primType] )
    PIDCounterS ++
else
    if ( PIDCounterR < R[primType] )
        PIDCounterR++
    else
        PrimitiveID++
        PIDCounterR = 0
endif
```
Two counters are employed to control the incrementing of PrimitiveID. The counters are compared against two corresponding parameters associated with the primitive topology type.

The PIDCounterS is used to 'skip over' some number (possibly zero) initial vertices of the primitive topology. This counter gets reset to 0 after each primitive is terminated.

Then the PIDCounterR is used to periodically increment the PrimitiveID, where the incrementing interval (vertex count) is topology-specific.

The following table lists the S[] and R[] values associated with each primitive topology type.

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>S, R</th>
<th>PrimitiveID Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTLIST</td>
<td>1, 0</td>
<td>0,1,2,3, ...</td>
</tr>
<tr>
<td>POINTLIST_BF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINELIST</td>
<td>1, 1</td>
<td>0,0,1,1,2,2,3,3, ...</td>
</tr>
<tr>
<td>LINELIST_ADJ</td>
<td>1, 3</td>
<td>0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3, ...</td>
</tr>
<tr>
<td>LINESTRIP</td>
<td>2, 0</td>
<td>0,0,1,2,3, ...</td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_ADJ</td>
<td>3, 0</td>
<td>0,0,1,2,3, ...</td>
</tr>
<tr>
<td>TRILIST</td>
<td>1, 2</td>
<td>0,0,0,1,1,2,2,2,3,3,3, ...</td>
</tr>
<tr>
<td>RECTLIST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRILIST_ADJ</td>
<td>1, 5</td>
<td>0,0,0,0,0,0,1,1,1,1,1,2,2,2,2,2,2,2, ...</td>
</tr>
<tr>
<td>TRISTRIP</td>
<td>3, 0</td>
<td>0,0,0,1,2,3, ...</td>
</tr>
<tr>
<td>TRISTRIP_REV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_ADJ</td>
<td>5, 1</td>
<td>0,0,0,0,0,0,1,1,2,2,3,3, ...</td>
</tr>
<tr>
<td>TRIFAN</td>
<td>3, 0</td>
<td>0,0,0,1,2,3, ...</td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUADLIST</td>
<td>1, 3</td>
<td>0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3, ...</td>
</tr>
<tr>
<td>Note: The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUADSTRIP</td>
<td>3, 1</td>
<td>0,0,0,0,1,1,2,2,3,3, ...</td>
</tr>
<tr>
<td>Note: The QUADSTRIP topology is converted to POLYGON topology at the beginning of the 3D pipeline.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**URBWrite**

The selected destination components are written into the destination VUE starting at **Destination Offset Select**. See the description of 3DPRIMITIVE for restrictions on this field.

**OutputBufferedVertex**

In order to accommodate 'cut' processing, the VF unit buffers one output vertex. The generation of a new vertex or the termination of a primitive causes the buffered vertex to be output to the pipeline.

**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).

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.

**Statistics Gathering**

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For LINELOOP, the last (closing) line object is not counted.</td>
</tr>
</tbody>
</table>
Vertex Shader (VS) Stage

VS Stage Overview

The VS stage of the 3D Pipeline is used to perform processing ("shading") of vertices after being 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 Vertex Cache to VS threads, and then pass the VS thread-generated vertices down the pipeline. Vertices that hit in the Vertex Cache are 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).

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 VS stage operation and control falls under these "common" functions; i.e., most stage state variables and VS 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 VS stage, and any exceptions the VS stage exhibits with respect to common FF unit functions.

State

URB_FENCE

Refer to 3D Overview for a description of how the VS stage processes this command.

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

Functions

The following pages describe the Vertex Shader 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:

<table>
<thead>
<tr>
<th>Project</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential indices are used in the 3DPRIMITIVE command (though this is effectively a don’t care as there would not be any VS$ hits).</td>
</tr>
</tbody>
</table>

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$.

<table>
<thead>
<tr>
<th>VS$ Function Enable</th>
<th>Mode of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLED (implicitly or explicitly)</td>
<td>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 ([Pre-DevHSW]) when the VF-generated vertices contain PrimitiveID. Otherwise the VS$ should be enabled.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>The VS$ is not used. VF stage assembles all vertices and writes them into the VUE supplied by the VS stage. VS stage subsequently spawns VS threads to process all vertices, overwriting the input data with the results. The VS stage pass references to these VUEs down the pipeline. Usage Model: This mode is only used when the VS function is required, but either (a) the VS kernel produces a side effect (e.g., writes to a memory buffer) which in turn requires every vertex to be processed by a VS thread, or (b) ([Pre-DevHSW]) the input vertex contains PrimitiveID.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>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$. In either case, the VS stage passes references to vertices (that hit or miss) down the pipeline without spawning any VS threads.</td>
</tr>
<tr>
<td>VS$</td>
<td>VS Function Enable</td>
</tr>
<tr>
<td>-----</td>
<td>--------------------</td>
</tr>
<tr>
<td></td>
<td>ENABLED</td>
</tr>
</tbody>
</table>

**SIMD4x2 VS Thread Request Generation**

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This section describes SIMD4x2 thread request generation, which is the only mode available.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This section describes SIMD4x2 thread execution, which is the only mode available.</td>
</tr>
</tbody>
</table>

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.

**Vertex Output**

VS threads must always write the destination URB handles passed in the payload. VS 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.

**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.

**Payloads**

The following pages describe the Vertex Shader Payloads.

**SIMD4x2 Payload**

The following table describes the payload delivered to VS threads.
### VS Thread Payload (SIMD4x2)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R0.6</td>
<td>31:24</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>23:0</td>
<td><strong>Thread ID.</strong> 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.</td>
</tr>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Offset:</strong> Specifies the extent of the scratch space allocated to the thread, specified as a 1KB-granular offset from the General State Base Address. See Scratch Space Base Offset description in VS_STATE. (See 3D Pipeline for further description on scratch space allocation). Format = GeneralStateOffset[31:10]</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td><strong>Project:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>8:0</td>
<td><strong>FFTID:</strong> 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range:</strong></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range</strong></td>
</tr>
<tr>
<td></td>
<td>9:0</td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td><strong>Binding Table Pointer.</strong> 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]</td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer.</strong> 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]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|       | 3:0  | **Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space).  
(See *3D Pipeline* for further description).  
Format = U4 power of two (in excess of 10)  
Range = [0,11] indicating [1K Bytes, 2M Bytes] |
| R0.2  | 31:0 | Reserved: delivered as zeros (reserved for message header fields) |
| R0.1  | 31:16| Reserved |
|       | 15:0 | **URB Return Handle 1:** This is the 64B-aligned URB offset where the EU’s upper channels (DWords 7:4) results are to be stored.  
If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread’s Channel Mask).  
(See *Generic FF Unit* for further description).  
Format:  
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
<tr>
<td>R0.0</td>
<td>31:16</td>
</tr>
</tbody>
</table>
|       | 15:0   | **URB Return Handle 0:** This is the 64B-aligned URB offset where the EU’s lower channels (DWords 3:0) results are to be stored.  
(See *Generic FF Unit* for further description).  
Format:  
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>
| [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. |
<p>| Varies  | 255:0 | <strong>Vertex Data:</strong> Data from (possibly) one or (more typically) two Vertex URB Entries is passed to the thread in the thread payload. The <strong>Vertex URB Entry Read Offset</strong> and <strong>Vertex URB Entry Read Length</strong> 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>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

3D Pipeline – Hull Shader (HS) Stage

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

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.

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.

**Dispatch Mask**

HS threads are dispatched with the dispatch mask set to 0xFFFF. It is the responsibility of the kernel to modify the execution mask as required (e.g., if operating in SIMD4x2 mode but only the lower half is active, as would happen in one thread if the threads were computing an odd number of OCPs via SIMD4x2 operation).

**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).

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>In SW Tessellation mode, the header contains <strong>Domain Point Count</strong> and <strong>Domain Point Buffer Starting Address</strong> fields which identify the domain points generated by an HS thread. The following tables show the fixed layouts of the Patch Header DW0-7, depending on DomainType and SW Tessellation Mode. <strong>HW Bug:</strong> 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:</td>
<td></td>
</tr>
<tr>
<td>QUAD domain</td>
<td></td>
</tr>
</tbody>
</table>
• 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:

```c
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];
}
```

### Patch Header (QUAD Domain)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31:0</td>
<td>UEQ0 Tessellation Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>6</td>
<td>31:0</td>
<td>VEQ0 Tessellation Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>5</td>
<td>31:0</td>
<td>UEQ1 Tessellation Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>4</td>
<td>31:0</td>
<td>VEQ1 Tessellation Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>3</td>
<td>31:0</td>
<td>Inside U Tessellation Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>2</td>
<td>31:0</td>
<td>Inside V Tessellation Factor</td>
</tr>
</tbody>
</table>
### Patch Header (TRI Domain)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31:0</td>
<td><strong>UEQ0 Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>6</td>
<td>31:0</td>
<td><strong>VEQ0 Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>5</td>
<td>31:0</td>
<td><strong>WEQ0 Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>4</td>
<td>31:0</td>
<td><strong>Inside Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>3-1</td>
<td>31:0</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>0</td>
<td>31:1</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Patch Header (ISOLINE Domain)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31:0</td>
<td><strong>Line Detail Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>6</td>
<td>31:0</td>
<td><strong>Line Density Tessellation Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: FLOAT32</td>
</tr>
<tr>
<td>5-0</td>
<td>31:0</td>
<td>Reserved : MBZ</td>
</tr>
</tbody>
</table>

### Patch Header (SW Tessellation Mode)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31:0</td>
<td><strong>Domain Point Count</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 &quot;culled&quot;, and the TE stage will discard the patch. Otherwise the TS stage will send this number of domain points down the pipeline. Format: U32</td>
</tr>
<tr>
<td>6</td>
<td>31:6</td>
<td><strong>Domain Point Buffer Starting Address (DPBSA)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <strong>Domain Point Count</strong> is 0. Format: 64B-aligned offset from SW Tessellation Base Address</td>
</tr>
<tr>
<td>5-0</td>
<td>31:0</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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    | 28:24| **PatchEnd**  
Set on the last domain point for the patch. By definition, PrimEnd must also be set.  
Programming Note: Software must ensure that the **Domain Point Count** 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.

ICP Dereferencing

If ICPs are only pushed in HS payloads (i.e., the Include Vertex Handles state bit is clear), the ICP handles are automatically released after the last instance for the patch is dispatched.

If Include Vertex Handles is set, the HS thread(s) will be reading ICP data in from the URB; it is the responsibility of the HS thread instances to explicitly dereference all the ICP handles via use of the Complete bit in URB_READ_xxx commands.

- If only one instance is used, that instance can dereference the ICP handles as soon as they are no longer needed, by setting Complete in the last URB_READ from that handle. Otherwise all (or the remaining) ICP handles need to be explicitly dereferenced via (possibly null-response-length) URB_READ commands prior to thread EOT.

- If more than one instance is spawned, the last-terminating instance is responsible for dereferencing all the ICP handles before it terminates. Instances can detect that they are the last-terminating thread via use of the semaphore allocated to the patch (via the Semaphore Handle and Semaphore Index payload fields). An URB_ATOMIC_INC operation (URB_ATOMIC command) can be performed on this semaphore by each instance prior to terminating. Only the last-terminating thread will observe the value (InstanceCount – 1) as a return value. After dereferencing all the ICPs, the last-terminating thread must also reset the semaphore to 0 via the URB_ATOMIC_MOV operation.
### 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

<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.6</td>
<td>31</td>
<td><strong>Dereference Thread</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:0</td>
<td>Thread ID. This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Form: Reserved for HW Implementation Use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Pointer</strong>. Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the General State Base Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = GeneralStateOffset[31:10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.5</td>
<td>9.0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.5</td>
<td>8.0</td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
<td><strong>Format</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
<td>U8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
<td><strong>Range</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
<td>0-255</td>
<td></td>
</tr>
<tr>
<td>GRF DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| 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).  
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 to be 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 | HSW     |          |
| 23:17     |      | **Instance Number**: A patch-relative instance number between 0 and InstanceCount-1.  
Format = U7 | HSW     |          |
| 16:13     |      | **Barrier Index**: This index is to be used in any BarrierMsgs sent by this thread to the Gateway.  
Format = U4 | HSW     |          |
| 12:0      |      | **Semaphore Handle**: This is the URB handle pointing to the first HS semaphore DWord in the URB. Software is responsible for statically allocating the semaphore Dwords in the URB. Refer to Semaphore Handle field in 3DSTATE_HS for size of semaphore allocation.  
Format: U12 64B-aligned URB Offset | HSW     |          |
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.1</td>
<td>31:0</td>
<td><strong>Primitive ID.</strong> This field contains the Primitive ID associated with the patch. Format: U32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.0</td>
<td>31:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Patch Data Record URB Return Handle. Format:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

R1 is only included for dispatches that have Include Vertex Handles enabled.

<p>| R1.7     | 31:16 | ICP 7 Handle ID         |         |          |
| R1.7     | 15:0  | ICP 7 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.6     | 31:16 | ICP 6 Handle ID         |         |          |
| R1.6     | 15:0  | ICP 6 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.5     | 31:16 | ICP 5 Handle ID         |         |          |
| R1.5     | 15:0  | ICP 5 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.4     | 31:16 | ICP 4 Handle ID         |         |          |
| R1.4     | 15:0  | ICP 4 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.3     | 31:16 | ICP 3 Handle ID         |         |          |
| R1.3     | 15:0  | ICP 3 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.2     | 31:16 | ICP 2 Handle ID         |         |          |
| R1.2     | 15:0  | ICP 2 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.1     | 31:16 | ICP 1 Handle ID         |         |          |
| R1.1     | 15:0  | ICP 1 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

<p>| R1.0     | 31:16 | ICP 0 Handle ID         |         |          |
| R1.0     | 15:0  | ICP 0 Handle            |         |          |</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>U13 64B-aligned URB offset.</td>
</tr>
</tbody>
</table>

R2 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count >7.

<p>| R2.7     | 31:16 | ICP 15 Handle ID        |         |          |
| R2.7     | 15:0  | ICP 15 Handle           |         |          |
| R2.6     | 31:16 | ICP 14 Handle ID        |         |          |
| R2.6     | 15:0  | ICP 14 Handle           |         |          |
| R2.5     | 31:16 | ICP 13 Handle ID        |         |          |
| R2.5     | 15:0  | ICP 13 Handle           |         |          |</p>
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2.4</td>
<td>15:0</td>
<td>ICP 13 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 12 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2.3</td>
<td>15:0</td>
<td>ICP 12 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 11 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2.2</td>
<td>15:0</td>
<td>ICP 11 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 10 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2.1</td>
<td>15:0</td>
<td>ICP 10 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 9 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2.0</td>
<td>15:0</td>
<td>ICP 9 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 8 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.7</td>
<td>15:0</td>
<td>ICP 8 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 23 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.6</td>
<td>15:0</td>
<td>ICP 22 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 22 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.5</td>
<td>15:0</td>
<td>ICP 21 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 21 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.4</td>
<td>15:0</td>
<td>ICP 20 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 20 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.3</td>
<td>15:0</td>
<td>ICP 19 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 19 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.2</td>
<td>15:0</td>
<td>ICP 18 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 18 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.1</td>
<td>15:0</td>
<td>ICP 17 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 17 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3.0</td>
<td>15:0</td>
<td>ICP 16 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 16 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.7</td>
<td>15:0</td>
<td>ICP 16 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 31 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.6</td>
<td>15:0</td>
<td>ICP 30 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 30 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.5</td>
<td>15:0</td>
<td>ICP 29 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 29 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.4</td>
<td>15:0</td>
<td>ICP 28 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:16</td>
<td>ICP 28 Handle ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R3 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count > 15

R4 is only included for dispatches that have Include Vertex Handles enabled and when ICP Count > 23
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4.3</td>
<td>31:16</td>
<td>ICP 27 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 27 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.2</td>
<td>31:16</td>
<td>ICP 26 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 26 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.1</td>
<td>31:16</td>
<td>ICP 25 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 25 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4.0</td>
<td>31:16</td>
<td>ICP 24 Handle ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 24 Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Varies]</td>
<td>255:0</td>
<td>Constant Data (optional):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optional</td>
<td></td>
<td>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).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Varies]</td>
<td>255:0</td>
<td>ICP Vertex Data (optional):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optional</td>
<td></td>
<td>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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HW Tessellation

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The TE stage can also operate in SW Tessellation mode, where it simply reads &quot;pre-tessellated&quot; domain point topologies from memory and passes them down the pipeline.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>In SW Tessellation mode, a Domain Point Count of 0 indicates that a patch is to be culled.</td>
</tr>
</tbody>
</table>

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: 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.
TRI Domain Tessellation

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

TRI Domain
**ISOLINE Domain Tessellation**

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

**ISOLINE Domain**
TF.LineDetail determines # segments

TF.LineDensity determines # lines

Line at V=1.0 not drawn
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**

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A DS kernel assumes it is to operate on two domain points in parallel using the EU’s SIMD4x2 execution model.</td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R0.6</td>
<td>31:24</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>23:0</td>
<td>Thread ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td>Scratch Space Offset. Specifies the offset of the scratch space allocated to the thread, specified as a 1KB-granular offset from the General State Base Address. See Scratch Space Base Offset description in VS_STATE. (See 3D Pipeline for further description on scratch space allocation). Format = GeneralStateOffset[31:10]</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Project:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>8:0</td>
<td></td>
<td>FFTID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td>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]</td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td>Sampler State Pointer. Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td><strong>Per Thread Scratch Space.</strong> Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]</td>
</tr>
<tr>
<td>R0.2</td>
<td>31:0</td>
<td>Reserved: delivered as zeros (reserved for message header fields)</td>
</tr>
<tr>
<td>R0.1</td>
<td>31:26</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>25:16</td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Handle ID 1.</strong> 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.</td>
</tr>
<tr>
<td>13:0</td>
<td></td>
<td><strong>URB Return Handle 1:</strong> 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>R0.0</td>
<td>31:26</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>25:16</td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Handle ID 0.</strong> 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>
<tr>
<td>15:14</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>13:0</td>
<td><strong>URB Return Handle 0:</strong> This is the URB handle where Vertex 0 data (the EU's lower channels (DWords 3:0)) results are to be stored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>R1.7</td>
<td>31:0</td>
<td><strong>PrimitiveID.</strong> 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.</td>
</tr>
<tr>
<td>R1.6</td>
<td>31:0</td>
<td><strong>Domain Point 1 W Coordinate.</strong> (See Domain Point 0 W Coordinate)</td>
</tr>
<tr>
<td>R1.5</td>
<td>31:0</td>
<td><strong>Domain Point 1 V Coordinate.</strong> (See Domain Point 0 V Coordinate)</td>
</tr>
<tr>
<td>R1.4</td>
<td>31:0</td>
<td><strong>Domain Point 1 U Coordinate.</strong> (See Domain Point 0 U Coordinate)</td>
</tr>
<tr>
<td>R1.3</td>
<td>31:14</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td></td>
<td>13:0</td>
<td><strong>Patch URB Handle.</strong> This is the URB handle of the Patch Record (common to both vertices).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>R1.2</td>
<td>31:0</td>
<td><strong>Domain Point 0 W Coordinate.</strong> If <strong>Compute W Coordinate Enable</strong> is set, this field will receive the computed value (1 – U – V) for Domain Point 0. Otherwise it is passed as 0.0.</td>
</tr>
<tr>
<td>R1.1</td>
<td>31:0</td>
<td><strong>Domain Point 0 V Coordinate.</strong> V coordinate associated with Domain Point 0.</td>
</tr>
<tr>
<td>R1.0</td>
<td>31:0</td>
<td><strong>Domain Point 0 U Coordinate.</strong> U coordinate associated with Domain Point 0.</td>
</tr>
<tr>
<td>Varies [Optional]</td>
<td>255:0</td>
<td><strong>Constant Data (optional).</strong></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Varies [Optional]</td>
<td>255:0</td>
<td><strong>Patch URB Data (optional)</strong>. 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.</td>
</tr>
</tbody>
</table>
3D Pipeline – 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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DSTATE_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_CONSTANT(Body)</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_PUSH_CONSTANT_ALLOC_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_BINDING_TABLE_POINTERS_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_SAMPLER_STATE_POINTERS_GS</td>
<td></td>
</tr>
<tr>
<td>3DSTATE_URB_GS</td>
<td></td>
</tr>
</tbody>
</table>

The state used by GS is defined with this inline state packet.
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

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Order of Vertices in Payload</th>
<th>GS Notes</th>
</tr>
</thead>
</table>
| <PRIMITIVE_TOPOLOGY> \( (N = \# \text{ of vertices}) \) | \[<\text{object#}>\) = \(<\text{vert#}>,...); \{\text{modified PrimType passed to thread}\} | \n
| POINTLIST | \[0\] = (0); \[1\] = (1); \ldots; \[N-2\] = (N-2); | \n
| POINTLIST_BF | N/A | \n
| LINELIST \( (N \text{ is multiple of 2}) \) | \[0\] = (0,1); \[1\] = (2,3); \ldots; \[(N/2)-1\] = (N-2,N-1) | \n
| LINELIST_ADJ \( (N \text{ is multiple of 4}) \) | \[0\] = (0,1,2,3); \[1\] = (4,5,6,7); \ldots; \[(N/4)-1\] = (N-4,N-3,N-2,N-1) | \n
| LINESTRIP \( (N \geq 2) \) | \[0\] = (0,1); | \n
<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Order of Vertices in Payload</th>
<th>GS Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;PRIMITIVE_TOPOLOGY&gt;</code></td>
<td><code>[&lt;object#&gt;] = (&lt;vert#&gt;,...); [modified PrimType passed to thread]</code></td>
<td></td>
</tr>
<tr>
<td>[1] = (1,2); ...;</td>
<td>[N-2] = (N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_ADJ (N &gt;= 4)</td>
<td>[0] = (0,1,2,3); [1] = (1,2,3,4); ...; [N-4] = (N-4,N-3,N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td>Same as LINESTRIP</td>
<td>Handled same as LINESTRIP</td>
</tr>
<tr>
<td>LINESTRIP_CONT_BF</td>
<td>Same as LINESTRIP</td>
<td>Handled same as LINESTRIP</td>
</tr>
<tr>
<td>LINELOOP (N &gt;= 2)</td>
<td>[0] = (0,1); [1] = (1,2); [N] = (N-1,0);</td>
<td>Not supported after GS.</td>
</tr>
<tr>
<td>TRILIST (N is multiple of 3)</td>
<td>[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>RECTLIST</td>
<td>Same as TRILIST</td>
<td>Handled same as TRILIST</td>
</tr>
<tr>
<td>TRILIST_ADJ (N is multiple of 6)</td>
<td>[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)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP (Reorder Leading) (N &gt;= 3)</td>
<td>[0] = (0,1,2); {TRISTRIP}; [1] = (1,3,2); {TRISTRIP_REV}; [k even] = (k,k+1,k+2) {TRISTRIP}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k odd] = (k,k+2,k+1) {TRISTRIP_REV}; [N-3] = (see above)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP (Reorder Trailing) (N &gt;= 3)</td>
<td>[0] = (0,1,2) {TRISTRIP}</td>
<td>&quot;Odd&quot; triangles have vertices reordered per OGL requirement, and identified as TRISTRIP_REV so the thread knows this.</td>
</tr>
<tr>
<td>PrimTopologyType</td>
<td>Order of Vertices in Payload</td>
<td>GS Notes</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;PRIMITIVE_TOPOLOGY&gt;</code></td>
<td>([&lt;object#&gt;] = (&lt;vert#&gt;,…); [modified PrimType passed to thread])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,1,3) (TRISTRIP_REV); ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k even] = (k,k+1,k+2) (TRISTRIP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k odd] = (k+1,k,k+2) (TRISTRIP_REV)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-3] = (see above)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_REV (Reorder Leading)</td>
<td>(N &gt;= 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (0,2,1) (TRISTRIP_REV); ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (1,2,3) (TRISTRIP); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k even] = (k,k+2,k+1) (TRISTRIP_REV)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k odd] = (k,k+1,k+2) (TRISTRIP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-3] = (see above)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_REV (Reorder Trailing)</td>
<td>(N &gt;= 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (1,0,2) (TRISTRIP_REV)</td>
<td>“Even” triangles have vertices reordered per OGL requirement, and identified as TRISTRIP_REV so the thread knows this.</td>
</tr>
<tr>
<td></td>
<td>[1] = (1,2,3) (TRISTRIP); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k even] = (k+1,k,k+2) (TRISTRIP_REV)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k odd] = (k,k+1,k+2) (TRISTRIP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-3] = (see above)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_ADJ (Reorder Leading)</td>
<td>(N &gt;= 6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N = 6 or 7:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (0,1,2,5,4,3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N = 8 or 9:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (0,1,2,6,4,3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,5,6,7,4,0); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N &gt;= 10:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (0,1,2,6,4,3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,5,6,8,4,0); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k&gt;1, even] = (2k,2k-2, 2k+2, 2k+6,2k+4, 2k+3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k&gt;2, odd] = (2k, 2k+3, 2k+4,</td>
<td></td>
</tr>
<tr>
<td>PrimTopologyType</td>
<td>Order of Vertices in Payload</td>
<td>GS Notes</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;PRIMITIVE_TOPOLOGY&gt;</code></td>
<td><code>[&lt;object#&gt;] = ([&lt;vert#&gt;],...); [modified PrimType passed to thread]</code></td>
<td></td>
</tr>
<tr>
<td><code>(N = # of vertices)</code></td>
<td>2k+6, 2k+2, 2k-2);...; Trailing object:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[(N/2)-3, even] = (N-6,N-8,N-4,N-1,N-2,N-3);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[(N/2)-3, odd] = (N-6,N-3,N-2,N-1,N-4,N-8);</code></td>
<td></td>
</tr>
<tr>
<td>TRISTRIP_ADJ (Reorder Trailing)</td>
<td>N = 6 or 7: [0] = (0,1,2,5,4,3)</td>
<td>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).</td>
</tr>
<tr>
<td><code>(N &gt;= 6)</code></td>
<td>N = 8 or 9: [0] = (0,1,2,6,4,3); [1] = (4,0,2,5,6,7); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N &gt; = 10: [0] = (0,1,2,6,4,3); [1] = (4,0,2,5,6,8); ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k&gt;1, even] = (2k,2k-2, 2k+2, 2k+6,2k+4, 2k+3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[k&gt;2, odd] = (2k+2, 2k-2, 2k, 2k+3, 2k+4, 2k+6);...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trailing object:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[(N/2)-3, even] = (N-6,N-8,N-4,N-1,N-2,N-3);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[(N/2)-3, odd] = (N-4,N-8,N-6,N-3,N-2,N-1);</code></td>
<td></td>
</tr>
<tr>
<td>TRIFAN (N &gt; 2)</td>
<td>[0] = (0,1,2); [1] = (0,2,3); ...; [N-3] = (0, N-2, N-1);</td>
<td>Only used by OGL</td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td>Same as TRIFAN</td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>Same as TRIFAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] = (0,1,2,3); [1] = (4,5,6,7); ...;</td>
<td>Not supported after GS. [DevSN8+]: QUADLIST primitives are converted into POLYGONS in VF, and therefore never reach</td>
</tr>
</tbody>
</table>
### Table: PrimTopologyType, Order of Vertices in Payload, GS Notes

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Order of Vertices in Payload</th>
<th>GS Notes</th>
</tr>
</thead>
</table>
| `<PRIMITIVE_TOPOLOGY>`  
(N = # of vertices) | `<object#> = (vert#,...); [modified PrimType passed to thread]` | the GS. |
|                  | [(N/4)-1] = (N-4,N-3,N-2,N-1); | Not supported after GS. |
|                  | [0] = (0,1,3,2);  
1 = (2,3,5,4); ...;  
[(N/2)-2] = (N-4,N-3,N-1,N-2); | [DevSNB+]: QUADSTRIP primitives are converted into POLYGONS in VF, and therefore never reach the GS. |
| Project: IVB+ | | |
| [DevIVB+]: 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) | similar to above |
| PATCHLIST_3..32 | | |

### 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 \(\rightarrow\) LINESTRIP, POLYGON \(\rightarrow\) TRIFAN, QUADs \(\rightarrow\) TRIFAN, etc.)

• Emulation of Polygon Fill Mode, where incoming polygons can be converted to points, lines (wireframe), or solid objects.

• Emulation of wide/sprite points.

When rendering is required, concurrent GS threads must use the FF_SYNC message (URB shared function) to request an initial VUE handle and synchronize output of VUEs to the pipeline (see URB in Shared Functions). Only one GS thread can be outputting VUEs to the pipeline at a time. To achieve parallelism, GS threads should perform the GS shader algorithm (along with any other required functions) and buffer results (either in the GRF or scratch memory) before issuing the FFSYNC message. The issuing GS thread is stalled on the FF_SYNC writeback until it is that thread’s turn to output VUEs. As only one GS thread at a time can output VUEs, the post-FF_SYNC output portion of the kernel should be optimized as much as possible to maximize parallelism.
Thread Execution

GS URB Entry

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

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

- When rendering is DISABLED, typically output vertices are 32B-aligned, with the exception of 16B-alignment for vertices <= 16B in length.
  - The absolute worst case size comes from three DW scalars output per vertex. If these are, say, three ".x" outputs, you need to store each DW in a 128b (16B) element, plus another pad 16B to keep the 32B alignment. So you require 4*16B = 64B/vertex. You have to have room for 1024 scalars / 3 scalar/vtx = 341 vertices. 341*64B = 21,824B. Then add 96B to hold 2b/vtx streamID and you get 21,920B entries.

- When rendering is ENABLED, each output vertex is 32B-aligned. Here the vertex header and vertex 'position' are required and therefore the minimum size vertex is 32B.
  - Here the worst case size isn't as bad as render-disabled, as you have to have a 4DW position output, plus any additional output. So, say you output 5 DW per vertex. You need 64B/vertex (16B vtx header, 16B position, 16B for the 2nd element, and 16B of pad). You have to have room for 1024 scalars / 5 = 204 vertices. 204*64B = 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 Output Topologies

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

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Supported for GS Thread Output?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINELIST</td>
<td>Yes</td>
</tr>
<tr>
<td>LINELIST_ADJ</td>
<td>No</td>
</tr>
<tr>
<td>LINESTRIP</td>
<td>Yes</td>
</tr>
<tr>
<td>LINESTRIP_ADJ</td>
<td>No</td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td>Yes</td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td>Yes</td>
</tr>
<tr>
<td>LINESTRIP_CONT_BF</td>
<td>Yes</td>
</tr>
<tr>
<td>LINELOOP</td>
<td>No</td>
</tr>
<tr>
<td>POINTLIST</td>
<td>Yes</td>
</tr>
<tr>
<td>POINTLIST_BF</td>
<td>Yes</td>
</tr>
<tr>
<td>POLYGON</td>
<td>Yes</td>
</tr>
<tr>
<td>QUADLIST</td>
<td>No</td>
</tr>
<tr>
<td>QUADSTRIP</td>
<td>No</td>
</tr>
<tr>
<td>RECTLIST</td>
<td>Yes</td>
</tr>
<tr>
<td>TRIFAN</td>
<td>Yes</td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td>Yes</td>
</tr>
<tr>
<td>TRILIST</td>
<td>Yes</td>
</tr>
<tr>
<td>TRILIST_ADJ</td>
<td>No</td>
</tr>
<tr>
<td>TRISTRIP</td>
<td>Yes</td>
</tr>
<tr>
<td>TRISTRIP_ADJ</td>
<td>No</td>
</tr>
<tr>
<td>TRISTRIP_REV</td>
<td>Yes</td>
</tr>
<tr>
<td>PATCHLIST_xxx</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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).

GS Invocations

Payloads

Thread Payload High-Level Layout

Thread Payload High-Level Layout shows the high-level layout of the payload delivered to GS threads.
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.

<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>30:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>R0.6</td>
<td>31</td>
<td><strong>Dereference Thread.</strong> This bit is defined to send back the Handle ID back to HS to dereference the input handles for this thread.</td>
</tr>
<tr>
<td></td>
<td>30:24</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>23:0</td>
<td><strong>Thread ID.</strong> 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.</td>
</tr>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Pointer.</strong> 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]</td>
</tr>
<tr>
<td></td>
<td>9:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>8:0</td>
<td><strong>FFTID.</strong> 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Range:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Project</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSW</td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td><strong>Binding Table Pointer:</strong> 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]</td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer.</strong> 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]</td>
</tr>
<tr>
<td><strong>GRF DWord</strong></td>
<td><strong>Bits</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3:0</td>
<td><strong>Per Thread Scratch Space.</strong> 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]</td>
<td></td>
</tr>
<tr>
<td>R0.2</td>
<td>31:24</td>
<td><strong>Semaphore Index.</strong> 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 &gt; 1. Format = U8</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td><strong>Hint.</strong> This is a copy of the corresponding 3DSTATE_GS bit. Format: U1</td>
</tr>
<tr>
<td></td>
<td>21:16</td>
<td><strong>Primitive Topology Type.</strong> 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 <strong>Discard Adjacency</strong> bit is set, the topology type passed in the payload is UNDEFINED. Format: See 3D Pipeline.</td>
</tr>
<tr>
<td></td>
<td>15:13</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>12:0</td>
<td><strong>Semaphore Handle.</strong> 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. Format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.1</td>
<td>31:27</td>
<td><strong>GS Instance ID 1.</strong> 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 &quot;dispatch mode&quot; is DUAL_OBJECT this field is not valid. Format: U5</td>
</tr>
<tr>
<td></td>
<td>26:16</td>
<td>Reserved.</td>
</tr>
<tr>
<td>GRF DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>R0.0</td>
<td>31:27</td>
<td><strong>GS Instance ID 0.</strong> 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 &quot;dispatch mode&quot; is DUAL_OBJECT, this field is not valid. Format: U5</td>
</tr>
<tr>
<td>26:16</td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>URB Return Handle 0.</strong> This is the URB offset where the EU's lower channels (DWords 3:0) results are stored. Format: HSW</td>
</tr>
</tbody>
</table>

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.

<p>| Rn.7       | 31:16 | ICP 7 Handle ID |
| Rn.6       | 31:16 | ICP 6 Handle ID |
| Rn.5       | 31:16 | ICP 5 Handle ID |</p>
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn.4</td>
<td>31:16</td>
<td>ICP 4 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 4 Handle</td>
</tr>
<tr>
<td>Rn.3</td>
<td>31:16</td>
<td>ICP 3 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 3 Handle</td>
</tr>
<tr>
<td>Rn.2</td>
<td>31:16</td>
<td>ICP 2 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 2 Handle</td>
</tr>
<tr>
<td>Rn.1</td>
<td>31:16</td>
<td>ICP 1 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 1 Handle</td>
</tr>
<tr>
<td>Rn.0</td>
<td>31:16</td>
<td>ICP 0 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 0 Handle</td>
</tr>
</tbody>
</table>

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.

<p>| Rn+2.7    | 31:16  | ICP 23 Handle ID    |
|           | 15:0   | ICP 23 Handle       |
| 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       |</p>
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn+2.4</td>
<td>31:16</td>
<td>ICP 20 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 20 Handle</td>
</tr>
<tr>
<td>Rn+2.3</td>
<td>31:16</td>
<td>ICP 19 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 19 Handle</td>
</tr>
<tr>
<td>Rn+2.2</td>
<td>31:16</td>
<td>ICP 18 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 18 Handle</td>
</tr>
<tr>
<td>Rn+2.1</td>
<td>31:16</td>
<td>ICP 17 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 17 Handle</td>
</tr>
<tr>
<td>Rn+2.0</td>
<td>31:16</td>
<td>ICP 16 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 16 Handle</td>
</tr>
<tr>
<td>Rn+3.7</td>
<td>31:16</td>
<td>ICP 31 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 31 Handle</td>
</tr>
<tr>
<td>Rn+3.6</td>
<td>31:16</td>
<td>ICP 30 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 30 Handle</td>
</tr>
<tr>
<td>Rn+3.5</td>
<td>31:16</td>
<td>ICP 29 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 29 Handle</td>
</tr>
<tr>
<td>Rn+3.4</td>
<td>31:16</td>
<td>ICP 28 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 28 Handle</td>
</tr>
<tr>
<td>Rn+3.3</td>
<td>31:16</td>
<td>ICP 27 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 27 Handle</td>
</tr>
<tr>
<td>Rn+3.2</td>
<td>31:16</td>
<td>ICP 26 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 26 Handle</td>
</tr>
<tr>
<td>Rn+3.1</td>
<td>31:16</td>
<td>ICP 25 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 25 Handle</td>
</tr>
<tr>
<td>Rn+3.0</td>
<td>31:16</td>
<td>ICP 24 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 24 Handle</td>
</tr>
</tbody>
</table>

The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and ICP Count > 23.

<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn+3.7</td>
<td>31:16</td>
<td>ICP 31 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 31 Handle</td>
</tr>
<tr>
<td>Rn+3.6</td>
<td>31:16</td>
<td>ICP 30 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 30 Handle</td>
</tr>
<tr>
<td>Rn+3.5</td>
<td>31:16</td>
<td>ICP 29 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 29 Handle</td>
</tr>
<tr>
<td>Rn+3.4</td>
<td>31:16</td>
<td>ICP 28 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 28 Handle</td>
</tr>
<tr>
<td>Rn+3.3</td>
<td>31:16</td>
<td>ICP 27 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 27 Handle</td>
</tr>
<tr>
<td>Rn+3.2</td>
<td>31:16</td>
<td>ICP 26 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 26 Handle</td>
</tr>
<tr>
<td>Rn+3.1</td>
<td>31:16</td>
<td>ICP 25 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 25 Handle</td>
</tr>
<tr>
<td>Rn+3.0</td>
<td>31:16</td>
<td>ICP 24 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>ICP 24 Handle</td>
</tr>
</tbody>
</table>

The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled.
<table>
<thead>
<tr>
<th>GRF</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn.4</td>
<td>31:16</td>
<td>Object 1 ICP 0 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 1 ICP 0 Handle</td>
</tr>
<tr>
<td>Rn.3</td>
<td>31:16</td>
<td>Object 0 ICP 3 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 3 Handle</td>
</tr>
<tr>
<td>Rn.2</td>
<td>31:16</td>
<td>Object 0 ICP 2 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 2 Handle</td>
</tr>
<tr>
<td>Rn.1</td>
<td>31:16</td>
<td>Object 0 ICP 1 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 1 Handle</td>
</tr>
<tr>
<td>Rn.0</td>
<td>31:16</td>
<td>Object 0 ICP 0 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 0 Handle</td>
</tr>
</tbody>
</table>

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.

<p>| 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 |
|            | 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     |</p>
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn+2.4</td>
<td>31:16</td>
<td>Object 1 ICP 8 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 1 ICP 8 Handle</td>
</tr>
<tr>
<td>Rn+2.3</td>
<td>31:16</td>
<td>Object 0 ICP 11 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 11 Handle</td>
</tr>
<tr>
<td>Rn+2.2</td>
<td>31:16</td>
<td>Object 0 ICP 10 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 10 Handle</td>
</tr>
<tr>
<td>Rn+2.1</td>
<td>31:16</td>
<td>Object 0 ICP 9 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 9 Handle</td>
</tr>
<tr>
<td>Rn+2.0</td>
<td>31:16</td>
<td>Object 0 ICP 8 Handle ID</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Object 0 ICP 8 Handle</td>
</tr>
</tbody>
</table>

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.

Varies 31:0 Pushed Vertex Data. There can be up to 32 vertices supplied, each with a size defined by the Vertex URB Entry Read Length state. The amount of data provided for each vertex is defined by...
<table>
<thead>
<tr>
<th>GRF DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>the <strong>Vertex URB Entry Read Length</strong> state. For SINGLE or DUAL_INSTANCE dispatch modes, the pushed data for Vertex 0 immediately follows any pushed constant data. The pushed data for Vertex 1 immediately follows Vertex 0, and so on. There is no upper/lower swizzling of data. For DUAL_OBJECT dispatch mode, the pushed vertex data is split into upper and lower halves with Object 0 input vertices in the lower half, and Object 1 input vertices in the upper half.</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Usage</th>
<th>FFs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel Start Pointer</strong></td>
<td>This field, together with the General State Pointer, specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State Pointer.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>GRF Register Block Count</strong></td>
<td>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.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>Single Program Flow (SPF)</strong></td>
<td>Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m &gt; 1). See CR0 description in ISA Execution Environment.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>Thread Dispatch Priority</strong></td>
<td>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).</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>Floating</strong></td>
<td>This determines the initial value of the Floating Point Mode bit of the EU's CR0</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>State Variable</td>
<td>Usage</td>
<td>FFs</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Point Mode</td>
<td>architectural register that controls floating point behavior in the EU core. (See ISA.)</td>
<td>threads</td>
</tr>
<tr>
<td>Exceptions Enable</td>
<td>This bitmask controls the exception handing logic in the EU. (See ISA.)</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>Sampler Count</td>
<td>This is a hint which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).</td>
<td>All stages supporting sampling (VS, GS, WM)</td>
</tr>
<tr>
<td>Binding Table Entry Count</td>
<td>This is a hint which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The notes included in Sampler Count (above) also apply to this field).</td>
<td>All FFs spawning threads</td>
</tr>
</tbody>
</table>

**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.

**NOTE:** 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)**

<table>
<thead>
<tr>
<th>Fixed Payload Header Field (non-FF-specific)</th>
<th>Description</th>
<th>FFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF Unit ID</td>
<td>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.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>Snapshot Flag</td>
<td></td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>Thread ID</td>
<td>This field uniquely identifies this thread within the FF unit over some period of time.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>Scratch Space Pointer</td>
<td>This is the starting location of the thread’s allocated scratch space, specified as an offset from the <strong>General State Base Address</strong>. Note that scratch space is allocated by the FF unit on a per-thread basis, based on the <strong>Scratch Space Base Pointer</strong> and <strong>Per-Thread Scratch Space Size</strong> 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.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td>Dispatch ID</td>
<td>This field identifies this thread within the outstanding threads spawned by the FF unit. This field does <em>not</em> uniquely identify the thread over any significant period of time.</td>
<td>All FFs spawning threads</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th><strong>Fixed Payload Header Field (non-FF-specific)</strong></th>
<th><strong>Description</strong></th>
<th><strong>FFs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binding Table Pointer</strong></td>
<td>This field, together with the <strong>Surface State Base Pointer</strong>, 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 <strong>Surface State Base Pointer</strong>. See <strong>Shared Functions</strong> for a description of a Binding Table.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>Sampler State Pointer</strong></td>
<td>This field, together with the <strong>General State Base Pointer</strong>, 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 <strong>General State Base Pointer</strong>. See <strong>Shared Functions</strong> for a description of a Sampler State Table.</td>
<td>All FFs spawning threads which sample (VS, GS, WM)</td>
</tr>
<tr>
<td><strong>Per Thread Scratch Space</strong></td>
<td>This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the <strong>Scratch Space Base Pointer</strong>, to ensure that the <strong>Maximum Number of Threads</strong> can each get <strong>Per-Thread Scratch Space</strong> size without exceeding the driver-allocated scratch space.</td>
<td>All FFs spawning threads</td>
</tr>
<tr>
<td><strong>Handle ID &lt;n&gt;</strong></td>
<td>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 <strong>URB_WRITE</strong> message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.</td>
<td>VS, GS, CLIP, SF</td>
</tr>
<tr>
<td><strong>URB Return Handle &lt;n&gt;</strong></td>
<td>This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.</td>
<td>VS, GS, CLIP, SF</td>
</tr>
</tbody>
</table>
Fixed Payload Header Field (non-FF-specific)

<table>
<thead>
<tr>
<th>Description</th>
<th>FFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Topology Type</td>
<td>As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (e.g., for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread. GEN4 kernels written to process different types of objects can use this value to direct that processing. E.g., when a CLIP kernel is to provide clipping for all the various primitive types, the kernel would need to examine the Primitive Topology Type to distinguish between point, lines, and triangle clipping requests. <strong>Note:</strong> 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. (E.g., certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).</td>
</tr>
</tbody>
</table>

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

**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 \( \min(128, 16 \times \text{GRF Register Block Count}) \). 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, its 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.
**3D Pipeline - Stream Output Logic (SOL) Stage**

The Stream Output Logic (SOL) stage receives 3D topologies originating in the VF 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 registers and commands for the 3D State Streamout.

**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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

**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**
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, convert 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 should not see incomplete objects: the VF will remove incomplete input objects, and the GS will remove GS-generated incomplete objects.

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.

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Order of Vertices Streamed Out</th>
<th>Any SOL Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;PRIMITIVE_TOPOLOGY</strong> &gt; (N = # of vertices)</td>
<td>[object#] = (vert#,...);</td>
<td></td>
</tr>
<tr>
<td>POINTLIST</td>
<td>[0] = (0);</td>
<td></td>
</tr>
<tr>
<td>POINTLIST_BF</td>
<td>[1] = (1);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-2] = (N-2);</td>
<td></td>
</tr>
<tr>
<td>LINELIST (N is multiple of 2)</td>
<td>[0] = (0,1);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(N/2)-1] = (N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>LINELIST_ADJ (N is multiple of 4)</td>
<td>[0] = (1,2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (5,6);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(N/4)-1] = (N-3,N-2)</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP</td>
<td>[0] = (0,1);</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td>[1] = (1,2);</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td>[N-2] = (N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT_BF (N &gt;= 2)</td>
<td>[0] = (1,2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-4] = (N-3,N-2)</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_ADJ (N &gt;= 4)</td>
<td>[0] = (1,2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] = (2,3);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[N-4] = (N-3,N-2)</td>
<td></td>
</tr>
<tr>
<td>PrimTopologyType</td>
<td>Order of Vertices Streamed Out</td>
<td>Any SOL Notes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LINELOOP</td>
<td>N/A</td>
<td>Not supported after VF.</td>
</tr>
<tr>
<td>TRILIST (N is multiple of 3)</td>
<td>[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)</td>
<td></td>
</tr>
<tr>
<td>RECTLIST</td>
<td>Same as TRILIST</td>
<td>Handled same as TRILIST.</td>
</tr>
<tr>
<td>TRILIST_ADJ (N is multiple of 6)</td>
<td>[0] = (0,2,4); [1] = (6,8,10); ...; [(N/6)-1] = (N-6,N-4,N-2)</td>
<td></td>
</tr>
<tr>
<td>TRISTRIP (N &gt;= 3)</td>
<td>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)</td>
<td>&quot;Odd&quot; triangles have vertices reordered to yield increasing leading vertices starting with v0.</td>
</tr>
<tr>
<td>TRISTRIP (N &gt;= 3)</td>
<td>REORDER_TRAILING [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)</td>
<td>&quot;Odd&quot; triangles have vertices reordered to yield increasing trailing vertices starting with v2.</td>
</tr>
<tr>
<td>TRISTRIP_REV (N &gt;= 3)</td>
<td>REORDER_LEADING [0] = (0,2,1) [1] = (1,3,2); ...; [k even] = (k,k+2,k+1) [k odd] = (k,k+1,k+2) [N-3] = (see above)</td>
<td>&quot;Even&quot; triangles have vertices reordered to yield increasing leading vertices starting with v0.</td>
</tr>
<tr>
<td>TRISTRIP_REV (N &gt;= 3)</td>
<td>REORDER_TRAILING [0] = (1,0,2) [1] = (1,3,2); ...; [k even] = (k+1,k,k+2) [k odd] = (k,k+1,k+2) [N-3] = (see above)</td>
<td>&quot;Even&quot; triangles have vertices reordered to yield increasing trailing vertices starting with v2.</td>
</tr>
<tr>
<td>TRISTRIP_ADJ (N even, N &gt;= 6)</td>
<td>REORDER_LEADING N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (2,6,4); ...; N &gt; 10: [0] = (0,2,4); [1] = (2,6,4); ...; [k&gt;1, even] = (2k, 2k+2, 2k+4); [k&gt;2, odd] = (2k, 2k+4, 2k+2);...;</td>
<td>&quot;Odd&quot; objects have vertices reordered to yield increasing-by-2 leading vertices starting with v0.</td>
</tr>
<tr>
<td>PrimTopologyType</td>
<td>Order of Vertices Streamed Out</td>
<td>Any SOL Notes</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>$[(N/2)\text{-}3, \text{even}] = (N-6,N-4,N-2);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[(N/2)\text{-}3, \text{odd}] = (N-6,N-2,N-4);$</td>
<td></td>
</tr>
<tr>
<td>TRISTRIPE_ADJ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(N even, N &gt;= 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REORDER_TRAILING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 6 or 7:</td>
<td>$[0] = (0,2,4)$</td>
<td></td>
</tr>
<tr>
<td>N = 8 or 9:</td>
<td>$[0] = (0,2,4);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[1] = (4,2,6);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>N &gt; 10:</td>
<td>$[0] = (0,2,4);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[1] = (4,2,6);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$[k&gt;1, \text{even}] = (2k, 2k+2, 2k+4);$</td>
<td>&quot;Odd&quot; objects have vertices reordered to yield increasing-by-2 trailing vertices starting with v4.</td>
<td></td>
</tr>
<tr>
<td>$[k&gt;2, \text{odd}] = (2k+2,2k, 2k+4,);...$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trailing object:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[(N/2)\text{-}3, \text{even}] = (N-6,N-4,N-2);$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[(N/2)\text{-}3, \text{odd}] = (N-4,N-6,N-2);$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIFAN (N &gt; 2)</td>
<td>$[0] = (0,1,2);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[1] = (0,2,3);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[N-3] = (0, N-2, N-1);$</td>
<td></td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td>Same as TRIFAN</td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>Same as TRIFAN</td>
<td></td>
</tr>
<tr>
<td>QUADLIST</td>
<td>N/A</td>
<td>Not supported after VF.</td>
</tr>
<tr>
<td>QUADSTRIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PATCHLIST_1</td>
<td>$[0] = (0);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[1] = (1);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[N-2] = (N-2);$</td>
<td></td>
</tr>
<tr>
<td>PATCHLIST_2</td>
<td>$[0] = (0,1);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[1] = (2,3);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[(N/2)\text{-}1] = (N-2,N-1)$</td>
<td></td>
</tr>
<tr>
<td>PATCHLIST_3..32</td>
<td>similar to above</td>
<td></td>
</tr>
</tbody>
</table>
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 SODECL lists of streams to determine which SO buffers are targeted. The Stream To Buffer Selects bits in 3DSTATE_SO_DECL_LIST must be programmed accordingly (if the buffer is targeted, the select bit must set, else it must be cleared).

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Surface Format Name</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32A32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32A32_SINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32A32_UINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32_SINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32B32_UINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32_FLOAT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32G32_SINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32_UINT</td>
<td></td>
</tr>
<tr>
<td>[DevSNB+]</td>
<td>R32_FLOAT</td>
<td></td>
</tr>
</tbody>
</table>

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

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 – General-Purpose Processing

Numerous state variable controls are provided to tailor the ClipTest function as required by the API or primitive characteristics. These controls allow a mode where all objects are passed to CLIP threads, and in this regard the CLIP stage can be used as a second GS stage. However, unlike the GS stage, primitives output by CLIP threads will not be subject to 3D Clipping, and therefore any clip-testing/clipping of these primitives (if required) would need to be performed by the CLIP thread itself.

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 processed by a thread (MustClip), discarded (TrivialReject) or passed down the pipeline unmodified (TrivialAccept). In the MustClip case, the spawned thread is responsible for performing the actual 3D Clipping algorithm. The CLIP thread 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 independent in that it is comprised of newly-generated VUEs, and does not share vertices with the source primitive or other CLIP-generated primitives.

New vertices produced by the CLIP threads are stored in the URB. Their Vertex Headers are then read from the VUEs in order to insert the relevant information into the 3D pipeline. The CLIP unit maintains the proper ordering of CLIP-generated primitives and any surrounding trivially-accepted primitives. The CLIP unit also supports multiple concurrent CLIP threads and maintains the proper ordering of the thread outputs as dictated by the order of the source objects.
The outgoing primitive stream is sent down the pipeline to the Strip/Fan (SF) FF stage (now including the read-back VUE Vertex Header data such as Vertex Position (NDC or screen space), RTAIndex, VPIndex, PointWidth) and control information (PrimType, PrimStart, PrimEnd) while the remainder of the vertex data remains in the VUE in the URB.

**Fixed Function Clipper**

The GPU supports Fixed Function Clipping.

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

**Concepts**

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

**The Clip Volume**

3D objects are optionally clipped to the *clip volume*. The clip volume is defined as the *intersection* of a set of *clip half-spaces*. Six of these half-spaces define the view volume, while additional, user-defined half-spaces can be employed to perform clipping (or at least culling) within the view volume. The CLIP stage design will permit the enable/disable of certain subsets of these clip half-spaces. This capability can be used, for example, to disable viewport, guardband, and near and far clipping as required by the API and other conditions.

**View Volume**

The intersection of the six view half-spaces defines the *view volume*. The view volume is defined in 4D clip space coordinates as:

<table>
<thead>
<tr>
<th>View Clip Plane</th>
<th>4D Clip Space</th>
<th>NDC space, positive w</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMIN (NDC Left)</td>
<td>clip.x &lt; -clip.w</td>
<td>ndc.x &lt; -1</td>
</tr>
<tr>
<td>XMAX (NDC Right)</td>
<td>clip.w &lt; clip.x</td>
<td>ndc.x &gt; 1</td>
</tr>
<tr>
<td>YMIN (NDC Bottom)</td>
<td>clip.y &lt; -clip.w</td>
<td>ndc.y &lt; -1</td>
</tr>
<tr>
<td>YMAX (NDC top)</td>
<td>clip.w &lt; clip.y</td>
<td>ndc.y &gt; 1</td>
</tr>
<tr>
<td>ZMIN (NDC Near)</td>
<td>OGL: clip.z &lt; -clip.w</td>
<td>OGL: ndc.z &lt; -1.0</td>
</tr>
</tbody>
</table>
Note that, since the 2D (X,Y) extent of the projected view volume is subsequently mapped to the 2D pixel space viewport, the terms \textit{viewport} and \textit{view volume} are used somewhat interchangeably in this discussion.

The CLIP unit will perform view volume clip test using NDC coordinates (the results of the speculative PerspectiveDivide). The treatment of negative ndc.w and invalid (NaN, +/-INF) coordinates is clarified below.

**Negative W Coordinates**

Consider for a moment vertices with a negative clip.w coordinate. Examination of the API definitions for \textit{outside} shows that it is impossible for that vertex to be considered inside both the XMIN (NDC Left) and XMAX (NDC Right) planes. The clip.x coordinate would need to be greater than or equal to some positive value (-clip.w) to be considered inside the XMIN plane, while also being less than or equal to the negative (clip.w) value to be considered inside the XMAX plane. Obviously both these conditions cannot be met simultaneously, so a vertex with a negative clip.w coordinate will always appear outside.

Surprisingly, it is possible for a vertex to be outside both the XMIN and XMAX planes (and likewise for the Y axis). This arises when clip.w is negative and clip.x falls between clip.w and -clip.w. Note, however, that in NDC space (post perspective-divide), this same vertex would be considered inside. This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. The CLIP stage will avoid this artifact by supporting an additional clip.w=0 clip plane – a negative ndc.rhw value indicates the point is outside of the clip.w=0 plane.

The assumption made in the Clip stage is that only the w>0 portion of clip space is considered visible. The VertexClipTest function tests each incoming 1/w value and, if negative, the vertex is tagged as being outside the w=0 plane. These vertex outcodes are combined in ClipDetermination to determine TA/TR/MC status.

A negative w coordinate poses an additional issue due to the fact that VertexClipTest is performed using post-perspection-projection coordinates (NDC or screen space). This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. For example, to test for (x>w) using NDC coordinates, (x/w>1) must be used when w>0, and (x/w<1) must be used when w<0. The VertexClipTest function therefore uses the sign of the incoming 1/w coordinate to select the appropriate comparison function for each of the VP and GB clip planes.

As the CLIP thread performs clipping in 4D clip space, only the truly visible portions of objects (i.e., meeting the 4D clip space visibility criteria) will be considered. The CLIP thread should not output negative w (clip or NDC) coordinates.
User-Specified Clipping

The various APIs define mechanisms by which objects can be clipped or culled according to some user-specified parameter(s) in addition to the implied viewport clipping. In GEN, the HW support of these mechanisms is restricted to use of the 8 UserClipFlags (UCFs) of the VUE Vertex Header. Software is required to provide the remaining support (e.g., the JITTER including GEN4 instructions to cause a distance value to be computed, tested for visibility, and generation of the appropriate UCF bit.)

Guard Band

Note: Refer to Vertex X,Y Clamping and Quantization in the SF stage section for device-specific guardband size information.

3D Clipping is time consuming. For cases where 2D Clipping is sufficient, we are willing to forgo 3D Clipping and instead apply 2D Clipping during rendering. In the general case, this is possible only when an object is totally within the ZMin and ZMax planes, and only clipping to the view volume X/Y MIN/MAX clip planes is required, as 2D Clipping is restricted to a screen-aligned 2D rectangle.
However, we must ensure that the 2D extent of these objects does not exceed the limitations of the renderer’s coordinate space (see Vertex X,Y Clamping and Quantization in the SF section). Therefore we define a 2D guardband region corresponding to (though likely somewhat smaller than) the maximum 2D extent supported by the renderer. During VertexClipTest, vertices are (optionally) subjected to an additional visibility test based on the 2D guardband region.

During ClipDetermination, if an object is not trivially-rejected from the 2D viewport, the XMIN_GB, XMAX_GB, YMIN_GB and YMAX_GB guardband outcodes are used instead of the XMIN, XMAX, YMIN, YMAX view volume outcodes to determine trivial-accept. This allows objects that fall within the guardband and possibly intersect the viewport to be trivially-accepted and passed down the pipeline.

The diagram below shows some examples of objects (triangles) in relation to the viewport and guardband. The shaded triangles are examples of triangles that are not trivially accepted to the viewport but trivially accepted to the guardband and therefore passed to down the pipeline. Without the guardband, these triangles would have to be submitted to a CLIP thread.

**Normal Guardband Operation**

The CLIP stage needs to handle the case where the viewport XY is larger than the screen space coordinate range supported by the SF and WM units. This condition may arise when the API defines an implicit 2D clip between the viewport XY extent and the render target. In the GEN4 3D pipeline, the guardband must be used to force explicit clipping in order to ensure legal coordinates are passed out of the CLIP stage. Therefore the CLIP unit supports a guardband that can be larger or smaller than the viewport (in any particular direction). The following diagram illustrates a case with a very large viewport,
extending well beyond the guardband. Note that the only trivial accept case is where objects are completely within the guardband.

Very Large Viewport Case

NDC Guardband Parameters

*Note: Refer to Vertex X,Y Clamping and Quantization in the SF stage section for device-specific guardband size information.*

When the CLIP unit performs VertexClipTest in NDC space, the guardband limits must be provided as NDC coordinates. The diagram below shows how the guardband NDC coordinates are derived. Specifically, the XMIN_GB NDC coordinate is simply the ratio of the (screen space) distance from the screen space VP center to the screen space GB XMin boundary over the distance from the VP center to the VP XMin (left) boundary. A similar computation yields the XMAX_GB (right), YMIN_GB (bottom) and YMAX_GB (top) guardband NDC coordinates.
As these guardband parameters are defined relative to the viewport, each of the up-to-16 sets of viewport specifications supported in the 3D pipeline will require a corresponding set of guardband parameters. These guardband parameters are provided as a separate memory-resident state structure (CLIP_VIEWPORT), and referenced via the **Clipper Viewport State Pointer** contained in the CLIP_STATE structure. Note that the CLIP_VIEWPORT structure has a different definition than the SF_VIEWPORT structure used by the SF unit.

**Vertex-Based Clip Testing Considerations**

The CLIP unit performs clip test and determines whether objects need to be clipped based solely on information (position, UserClipFlags) provided at the vertices of the object as they arrive at the clip stage. Issues arise if and when the corresponding rendered object is not constrained to the convex hull of the object. Different APIs impose different treatment of these conditions.

In addition and in the more general case, a CLIP thread could be used to convert the object (as defined by its vertices) into some arbitrary output primitive. In this case, the CLIP unit’s ClipTest/ClipDetermination logic may not be suitable for determination of when to reject/accept/clip objects. In this case the ClipMode can be used to route all (or all non-rejected) objects to CLIP threads, where the proper clip-test and clipping can occur in the CLIP kernel.

One issue that arises is whether a trivial-reject to the VPXY is suitable. If this were allowed, an object might be discarded even if it would have been partially visible in the viewport. A second issue is whether a TA against the GB is suitable. If this were allowed, portions of the rendered object might be visible in the VP even if the object should have been clipped out of the VP.

**Triangle Objects**

In the normal processing of triangle-based primitives (tristrip/trilist/polygon/etc.), the footprint of each triangle is constrained to the 2D convex hull. I.e., the rendering of these triangles will not produce pixels outside of the triangle. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for triangle objects:

- Both the VPXY and GB clip boundaries can be utilized (as described above). If the triangle is TR against the VP, it can be discarded. Otherwise, if the triangle is TA against the GB, it can be passed down the pipeline (assuming it is TA against VPZ, UCFs, etc.) and properly handled by 2DClipping.
- The GB parameters can be programmed to coincide with the maximum allowable screen space extent (though making the GB marginally smaller than this max extent is highly recommended).

**Non-Wide Line Objects**

In the normal processing of non-wide, line-based primitives (linestrip/linelist/etc.), the footprint of each line is constrained to the 2D convex hull. I.e., the rendering of these lines will not produce pixels off of the line. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for non-wide line objects. (See Triangle Objects above).

**Wide Line Objects**

The GEN rendering hardware supports wide lines (solid lines with a line width or anti-aliased lines). When rendered, pixels outside of the convex hull will be generated.

The following diagram shows an example of a wide line that normally would be TA against the GB. If the TA is allowed, the partially-visible region of the line would be rendered.

In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the line must be clipped-out against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide lines.

**Wide Points**

The GEN rendering hardware supports a width parameter for native line objects. When rendered, pixels surrounding the point (center) vertex will be generated.

The following diagram shows an example wide point that normally would be TR against the VPXY. If the TR is allowed, the partially-visible region of the point would not be rendered.
In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the point must be TR against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide points.

**RECTLIST**

The CLIP unit treats RECTLIST exactly like TRILIST. No special consideration is made for the implied 4th vertex of each rectangle (although ViewportXY and Guardband VertexClipTest theoretically should be sufficient to drive ClipDetermination). Given this, and the fact that RECTLIST is primarily intended for driver-generated *BLT* functions, there are number of restrictions on the use of RECTLIST, especially regarding the CLIP unit. Refer to the RECTLIST definition in 3D Pipeline.

**3D Clipping**

If an object needs to be clipped, it is passed to the CLIP thread. The CLIP thread performs some (arbitrary) algorithm to clip the primitive, and subsequently output new vertices as a primitive defining the visible region of the input object (assuming there is a visible region). In the process of spawning the CLIP thread, the input vertices may be considered *consumed* and therefore dereferenced. Therefore the CLIP thread needs to copy (if required) any input VUE data to a new output VUE; there is no mechanism to *output* input vertices other than copying.

[DevSNB+] supports only Fixed function Clipping.

**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

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:

```c
//
// 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
}
```
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 (DevBW-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_XMAX)
        }
    #endif

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

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

#ifdef (DevBW-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)
#endif (DevBW-E0)
if (rhw_neg) {
    outCode[GB_XMIN] = (x >= gb_XMIN)
    outCode[GB_XMAX] = (x <= gb_XMAX)
    outCode[GB_YMIN] = (y >= gb_YMIN)
    outCode[GB_YMAX] = (y <= gb_YMAX)
}
#endif
if (rhw_neg) {
    outCode[GB_XMIN] = (x > gb_XMIN)
    outCode[GB_XMAX] = (x < gb_XMAX)
    outCode[GB_YMIN] = (y > gb_YMIN)
    outCode[GB_YMAX] = (y < gb_YMAX)
}
}
}

// Handle case where either VP or GB disabled (but not both)
// In this case, the disabled set take on the outcodes of the enabled set
//
if (CLIP_STATE.ViewportXYClipTestEnable && !CLIP_STATE.GuardbandClipTestEnable) {
    outCode[GB_XMIN] = outCode[VP_XMIN]
    outCode[GB_XMAX] = outCode[VP_XMAX]
    outCode[GB_YMIN] = outCode[VP_YMIN]
    outCode[GB_YMAX] = outCode[VP_YMAX]
} else if (!CLIP_STATE.ViewportXYClipTestEnable && CLIP_STATE.GuardbandClipTestEnable) {
    outCode[VP_XMIN] = outCode[GB_XMIN]
    outCode[VP_XMAX] = outCode[GB_XMAX]
    outCode[VP_YMIN] = outCode[GB_YMIN]
    outCode[VP_YMAX] = outCode[GB_YMAX]
}
//
// X/Y/Z NaN Handling
//
xyorgben = (CLIP_STATE.ViewportXYClipTestEnable ||
CLIP_STATE.GuardbandClipTestEnable)
if (isNAN(x)) {
    outCode[GB_XMIN] = xyorgben
    outCode[GB_XMAX] = xyorgben
    outCode[VP_XMIN] = xyorgben
    outCode[VP_XMAX] = xyorgben
}
if (isNAN(y)) {
    outCode[GB_YMIN] = xyorgben
    outCode[GB_YMAX] = xyorgben
    outCode[VP_YMIN] = xyorgben
    outCode[VP_YMAX] = xyorgben
}
if (isNaN) {
    outCode[VP_ZMIN] = CLIP_STATE.ViewportZClipTestEnable
    outCode[VP_ZMAX] = CLIP_STATE.ViewportZClipTestEnable
}

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

**Object Staging**

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

**Partial Object Removal**

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

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

**ClipDetermination Function**

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

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

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

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

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way, objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

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

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

```
//
```
// ClipDetermination

// Compute objANDCode and objORCode

switch (object type) {
    case POINT:
    {
        objANDCode[...] = v0.outCode[...]
        objORCode[...] = v0.outCode[...]
    } break
    case LINE:
    {
        objANDCode[...] = v0.outCode[...] & v1.outCode[...]
        objORCode[...] = v0.outCode[...] | v1.outCode[...]
    } break
    case TRIANGLE:
    {
        objANDCode[...] = v0.outCode[...] & v1.outCode[...] & v2.outCode[...]
        objORCode[...] = v0.outCode[...] | v1.outCode[...] | v2.outCode[...]
    } break

    // Determine TR/TA against interesting boundary subsets
    //
    TR_VPXY = (objANDCode[VP_L] | objANDCode[VP_R] | objANDCode[VP_T] | objANDCode[VP_B])
    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

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 (i.e., 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).

```c
//
// 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: {
```

555
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.

<table>
<thead>
<tr>
<th>Input PrimTopologyType</th>
<th>Pass-Through Output PrimTopologyType</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTLIST</td>
<td>POINTLIST</td>
<td></td>
</tr>
<tr>
<td>POINTLIST_BF</td>
<td>POINTLIST_BF</td>
<td></td>
</tr>
<tr>
<td>LINELIST</td>
<td>LINELIST</td>
<td></td>
</tr>
<tr>
<td>LINELIST_ADJ</td>
<td>LINELIST</td>
<td>Adjacent vertices removed.</td>
</tr>
<tr>
<td>LINESTRIP</td>
<td>LINESTRIP</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_ADJ</td>
<td>LINESTRIP</td>
<td>Adjacent vertices removed.</td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td>LINESTRIP_BF</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td>LINESTRIP_CONT</td>
<td></td>
</tr>
<tr>
<td>LINESTRIP_CONT_BF</td>
<td>LINESTRIP_CONT_BF</td>
<td></td>
</tr>
<tr>
<td>LINELOOP</td>
<td>N/A</td>
<td>Not supported after GS.</td>
</tr>
<tr>
<td>TRILIST</td>
<td>TRILIST</td>
<td></td>
</tr>
<tr>
<td>RECTLIST</td>
<td>RECTLIST</td>
<td></td>
</tr>
<tr>
<td>TRILIST_ADJ</td>
<td>TRILIST</td>
<td>Adjacent vertices removed.</td>
</tr>
<tr>
<td>TRISTRIP</td>
<td>TRISTRIP or TRISTRIP_REV</td>
<td>Depends on where the incoming strip is broken (if at all) by discarded or clipped objects. See Tristrip Clipping Notes subsection.</td>
</tr>
<tr>
<td>TRISTRIP_REV</td>
<td>TRISTRIP or TRISTRIP_REV</td>
<td>Depends on where the incoming strip is broken (if at all) by discarded or clipped objects. See Tristrip Clipping Notes subsection.</td>
</tr>
<tr>
<td>TRISTRIP_ADJ</td>
<td>TRISTRIP or TRISTRIP_REV</td>
<td>Depends on where the incoming strip is broken (if at all) by discarded or clipped objects. Adjacent vertices removed. See Tristrip Clipping Notes subsection.</td>
</tr>
<tr>
<td>TRIFAN</td>
<td>TRIFAN</td>
<td></td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td>TRIFAN_NOSTIPPLE</td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>POLYGON</td>
<td></td>
</tr>
<tr>
<td>QUADLIST</td>
<td>N/A</td>
<td>Not supported after GS.</td>
</tr>
<tr>
<td>QUADSTRIP</td>
<td>N/A</td>
<td>Not supported after GS.</td>
</tr>
</tbody>
</table>
**Primitive Output**

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

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

**Other Functionality**

**Statistics Gathering**

The CLIP unit includes logic to assist in the gathering of certain pipeline statistics. The statistics take the form of MI counter registers (see Memory Interface Registers), where the CLIP unit provides signals causing those counters to increment.

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

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

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

**CL_INVOCATION_COUNT**

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

To maintain a count of application-generated objects, software must clear the CLIP unit’s Statistic Enable whenever driver-generated objects are rendered.
3D Pipeline - Strips and Fans (SF) Stage

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

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

Inputs from CLIP

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

SF's Vertex Pipeline Inputs

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>primType</td>
<td>enum</td>
<td>Type of primitive topology the vertex belongs to.  <em>Primitive Assembly</em> for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a list of primitive types supported by the SF unit. See <em>3D Pipeline</em> for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptions of these topologies.</td>
</tr>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CLIP unit will convert any primitive with adjacency (3DPRIMxxx_ADJ) it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>receives from the pipeline into the corresponding primitive without</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adjacency (3DPRIMxxx).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software must use a GS thread to convert these to some other (supported)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>primitive type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DevSNB+] If an object is clipped by the hardware clipper, the CLunit would</td>
</tr>
<tr>
<td></td>
<td></td>
<td>force this field to <em>3DPRIM POLYGON</em>. SFunit would process this incoming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>object just as it would any other <em>3DPRIM POLYGON</em>. SFunit selects vertex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 as the provoking vertex.</td>
</tr>
<tr>
<td>primStart,primEnd</td>
<td>boolean</td>
<td>Indicate vertex’s position within the primitive topology</td>
</tr>
<tr>
<td>vInX[]</td>
<td>float</td>
<td>Vertex X position (screen space or NDC space)</td>
</tr>
<tr>
<td>vInY[]</td>
<td>float</td>
<td>Vertex Y position (screen space or NDC space)</td>
</tr>
<tr>
<td>vInZ[]</td>
<td>float</td>
<td>Vertex Z position (screen space or NDC space)</td>
</tr>
<tr>
<td>vInInvW[]</td>
<td>float</td>
<td>Reciprocal of Vertex homogeneous (clip space) W</td>
</tr>
<tr>
<td>hVUE[]</td>
<td>URB</td>
<td>Points to the vertex’s data stored in the URB (one VUE handle per vertex)</td>
</tr>
<tr>
<td>renderTargetArrayIndex</td>
<td>uint</td>
<td>Index of the render target (array element or 3D slice), clamped to 0 by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GS unit if the max value was exceeded.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this vertex is the leading vertex of an object within the primitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>topology, this value will be associated with that object in subsequent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>processing.</td>
</tr>
<tr>
<td>viewportIndex</td>
<td>uint</td>
<td>Index of a viewport transform matrix within the SF_VIEWPORT structure used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to transform the vertex data.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>perform</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>pointSize</td>
<td>uint</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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, RTIndex 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**

<table>
<thead>
<tr>
<th>primType</th>
<th>VertexCount Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DPRIM_TRILIST</td>
<td>nonzero multiple of 3</td>
</tr>
<tr>
<td>3DPRIM_TRISTRIP</td>
<td>&gt;=3</td>
</tr>
<tr>
<td>3DPRIM_TRISTRIP_REVERSE</td>
<td>&gt;=3</td>
</tr>
<tr>
<td>3DPRIM_TRIFAN</td>
<td>&gt;=3</td>
</tr>
<tr>
<td>3DPRIM_TRIFAN_NOSTIPPLE</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_POLYGON</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_LINELIST</td>
<td>nonzero multiple of 2</td>
</tr>
<tr>
<td>3DPRIM_LINELIST</td>
<td>&gt;=2</td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP_CONT</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_LINESTRIP_CONT_BF</td>
<td></td>
</tr>
<tr>
<td>3DPRIM_RECTLIST</td>
<td>nonzero multiple of 3</td>
</tr>
<tr>
<td>3DPRIM_POINTLIST</td>
<td>nonzero</td>
</tr>
</tbody>
</table>

*Primitive Assembly* for a list of the 3D object types.

**3D Object Types**

<table>
<thead>
<tr>
<th>objectType</th>
<th>generated by primType</th>
<th>Vertices/Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DOBJ_POINT</td>
<td>3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF</td>
<td>1</td>
</tr>
<tr>
<td>3DOBJ_LINE</td>
<td>3DPRIM_LINELIST 3DPRIM_LINELIST 3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_CONT_BF</td>
<td>2</td>
</tr>
<tr>
<td>3DOBJ_TRIANGLE</td>
<td>3DPRIM_TRILIST 3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE 3DPRIM_TRIFAN</td>
<td>3</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>objectType</td>
<td>enum</td>
<td>Type of object. <em>Primitive Assembly</em></td>
</tr>
<tr>
<td>nV</td>
<td>uint</td>
<td>The number of object vertices passed to Object Setup. <em>Primitive Assembly</em></td>
</tr>
<tr>
<td>v[0..nV-1]*</td>
<td>various</td>
<td>Data arrays associated with object vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. <em>Primitive Assembly</em></td>
</tr>
<tr>
<td>invertOrientation</td>
<td>enum</td>
<td>Indicates whether the orientation (CW or CCW winding order) of the vertices of a triangle object should be inverted. Ignored for non-triangle objects.</td>
</tr>
<tr>
<td>backFacing</td>
<td>enum</td>
<td>Valid only for points and line objects, indicates a back facing object. This is used later for culling.</td>
</tr>
<tr>
<td>provokingVtx</td>
<td>uint</td>
<td>Specifies the index (into the 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.)</td>
</tr>
<tr>
<td>polyStippleEnable</td>
<td>boolean</td>
<td>TRUE if Polygon Stippling is enabled. FALSE for TRIFAN_NOSTIPPLE. Ignored for non-triangle objects.</td>
</tr>
<tr>
<td>continueStipple</td>
<td>boolean</td>
<td>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.</td>
</tr>
<tr>
<td>renderTargetIndex</td>
<td>uint</td>
<td>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.</td>
</tr>
<tr>
<td>viewPortIndex</td>
<td>uint</td>
<td>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.</td>
</tr>
<tr>
<td>pointSize</td>
<td>unit</td>
<td>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.</td>
</tr>
</tbody>
</table>
The following table defines, for each primitive topology type, which vertex’s VPIndex/RTAIndex applies to the objects within the topology.

### VPIndex/RTAIndex Selection

<table>
<thead>
<tr>
<th>PrimTopologyType</th>
<th>Viewport Index Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTLIST</td>
<td>Each vertex supplies the VPIndex for the corresponding point object</td>
</tr>
<tr>
<td>POINTLIST_BF</td>
<td></td>
</tr>
<tr>
<td>LINELIST</td>
<td>The leading vertex of each line supplies the VPIndex for the corresponding line object.</td>
</tr>
<tr>
<td></td>
<td>V0.VPIndex → Line(V0,V1)</td>
</tr>
<tr>
<td></td>
<td>V2.VPIndex → Line(V2,V3)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>LINESTRIP</td>
<td>The leading vertex of each line segment supplies the VPIndex for the corresponding line object.</td>
</tr>
<tr>
<td>LINESTRIP_BF</td>
<td>V0.VPIndex → Line(V0,V1)</td>
</tr>
<tr>
<td>LINESTRIP_CONT</td>
<td>V1.VPIndex → Line(V1,V2)</td>
</tr>
<tr>
<td>LINESTRIP_CONT_BF</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> If the VPIndex changes within the topology, shared vertices will be processed (mapped) multiple times.</td>
</tr>
<tr>
<td>RECTLIST</td>
<td></td>
</tr>
<tr>
<td>TRILIST</td>
<td>The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects.</td>
</tr>
<tr>
<td>RECTLIST</td>
<td>V0.VPIndex → Tri(V0,V1,V2)</td>
</tr>
<tr>
<td></td>
<td>V3.VPIndex → Tri(V3,V4,V5)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>TRISTRIP</td>
<td>The leading vertex of each triangle supplies the VPIndex for the corresponding triangle object.</td>
</tr>
<tr>
<td>TRISTRIP_REVERSE</td>
<td>V0.VPIndex → Tri(V0,V1,V2)</td>
</tr>
<tr>
<td></td>
<td>V1.VPIndex → Tri(V1,V2,V3)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> If the VPIndex changes within the primitive, shared vertices will be processed (mapped) multiple times.</td>
</tr>
<tr>
<td>TRIFAN</td>
<td></td>
</tr>
<tr>
<td>TRIFAN_NOSTIPPLE</td>
<td>The first vertex (V0) supplies the VPIndex for all triangle objects.</td>
</tr>
<tr>
<td>POLYGON</td>
<td></td>
</tr>
</tbody>
</table>
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 I in [0..vertexCount-1]) {
        v[0] ← vIn[i] // copy all arrays (e.g., vX, vY, etc.)
    }
    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.

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

for each (vertex I in [0..vertexCount-1]) {
    v[0] arrays ← vIn[i] arrays
    v[1] arrays ← vIn[i+1] arrays
    ObjectSetup()
    continueStipple = TRUE
}

Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

3DPRIM_TRILIST Primitive

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

TriangleListDecomposition() {
    objectType = 3DOBJ_TRIANGLE
    nV = 3
    invertOrientation = FALSE
}
provokingVtx = **Triangle List/Strip Provoking Vertex Select**

polyStippleEnable = TRUE

for each (vertex I in [0..vertexCount-3] by 3) {
  v[0] arrays ← vIn[i] arrays
  v[1] arrays ← vIn[i+1] arrays
  v[2] arrays ← vIn[i+2] arrays
  ObjectSetup()
}
}

**Triangle Strip Decomposition**

The 3DPRIM_TRISTRIP and 3DPRIM_TRISTRIP_REVERSE primitives specify a series of triangles arranged in a strip, as illustrated below.

**3DPRIM_TRISTRIP[.REVERSE] Primitive**

![Diagram of a triangle strip]

The decomposition process divides the strip into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v1,v2,v3; v2,v3,v4; and so on. Note that the **winding order** of the vertices alternates between CW (clockwise), CCW (counter-clockwise), CW, etc. The **provokingVertex** of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

The 3D pipeline uses the winding order of the vertices to distinguish between front-facing and back-facing triangles (**Triangle Orientation (Face) Culling** below). Therefore, the 3D pipeline must account for the alternation of winding order in strip triangles. The **invertOrientation** variable is generated and used for this purpose.
To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (e.g., due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

```c
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.
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 \textit{invertOrientation} variable is output as FALSE unconditionally. The \textit{provokingVertex} of each object is taken from the \textbf{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 \textit{polyStippleEnable} variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```c
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:

\[
\begin{align*}
    x' &= m_{00} \cdot x + m_{30} \\
    y' &= m_{11} \cdot y + m_{31} \\
    z' &= m_{22} \cdot z + m_{32}
\end{align*}
\]

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

Destination Origin Bias

The positioning of the pixel sampling grid is programmable and is controlled by the Destination Origin Horizontal/Vertical Bias state variables (set via SF_STATE). If these bias values are both 0, pixels are sampled on an integer grid. Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0,0) falls within the primitive.

If the bias values are both 0.5, pixels are sampled on a half integer grid (i.e., X.5, Y.5). Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0.5,0.5) falls within the primitive. This positioning of the sample grid corresponds with the OpenGL rasterization rules, where fragment centers lay on a half-integer grid. It also corresponds with the Intel740 rasterizer (though that device did not employ top left rules).
Note that subsequent descriptions of rasterization rules for the various objects will be with reference to the pixel sampling grid.

**Destination Origin Bias**

**Point Rasterization Rule Adjustment**

POINT objects are rasterized as square RECTANGLEs, with one exception: The **Point Rasterization Rule** state variable (in SF_STATE) controls the rendering of point object edges that fall directly on pixel sample points, as the treatment of these edge pixels varies between APIs.

**RASTRULE_UPPER_LEFT**
Drawing Rectangle Offset Application

The Drawing Rectangle Offset subfunction offsets the object’s vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window’s or viewport’s upper-left corner.

Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the *clipped* draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.
Onscreen Draw Rectangle

Partially-offscreen Draw Rectangle

3DSTATE_DRAWING_RECTANGLE

Point Width Application

This stage of the pipeline applies only to 3DOBJ_POINT objects. Here the point object is converted from a single vertex to four vertices located at the corners of a square centered at the point’s X,Y position. The width and height of the square are specified by a point width parameter. The Point Width Source value in SF_STATE determines the source of the point width parameter: the point width is either taken from the Point Width value programmed in SF_STATE or the PointWidth specified with the vertex (as read back from the vertex VUE earlier in the pipeline).

The corner vertices are computed by adding and subtracting one half of the point width. Point Width Application.
**Point Width Application**

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

**Rectangle Completion**

This stage of the pipeline applies only to 3DOBJ_RECTANGLE objects. Here the X,Y coordinates of the 4th (upper right) vertex of the rectangle object is computed from the first 3 vertices as shown in the following diagram. The other vertex attributes assigned to the implied vertex (v[3]) are UNDEFINED as they are not used. The Object Setup subfunction will use the values at only the first 3 vertices to compute attribute interpolants used across the entire rectangle.

**Rectangle Completion**

\[ \text{Implied Vertex} = v2 + v0 - v1 \]
### 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:

<table>
<thead>
<tr>
<th>Project</th>
<th>Supported X,Y ScreenSpace &quot;Guardband&quot; Extent</th>
<th>Maximum Post-Clamp Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>([-32K,32K-1])</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>objType</th>
<th>Degenerate Object Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DOBJ_POINT</td>
<td>Two or more corner vertices are coincident (i.e., the radius quantized to zero)</td>
</tr>
<tr>
<td>3DOBJ_LINE</td>
<td>The endpoints are coincident</td>
</tr>
<tr>
<td>3DOBJ_TRIANGLE</td>
<td>All three vertices are collinear or any two vertices are coincident and SOLID fill mode applies to the triangle</td>
</tr>
<tr>
<td>3DOBJ_RECTANGLE</td>
<td>Two or more corner vertices are coincident</td>
</tr>
</tbody>
</table>
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

![Triangle Winding Order Diagram]

The *Front Winding* state variable in SF_STATE controls whether CW or CCW triangles are considered as having a *front-facing* orientation (at which point non-front-facing triangles are considered *back-facing*). The internal variable *invertOrientation* associated with the triangle object is then used to determine whether the orientation of a that triangle should be inverted. Recall that this variable is set in the Primitive Decomposition stage to account for the alternating orientations of triangles in strip primitives resulting form the ordering of the vertices used to process them.

The *Cull Mode* state variable in SF_STATE specifies how triangles are discarded according to their resultant orientation. See *Degenerate Objects*.

### Cull Mode

<table>
<thead>
<tr>
<th>CullMode</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CULLMODE_NONE</td>
<td>The face culling operation is disabled.</td>
</tr>
<tr>
<td>CULLMODE_FRONT</td>
<td>Triangles with <em>front facing</em> orientation are discarded.</td>
</tr>
<tr>
<td>CULLMODE_BACK</td>
<td>Triangles with <em>back facing</em> orientation are discarded.</td>
</tr>
<tr>
<td>CULLMODE_BOTH</td>
<td>All triangles are discarded.</td>
</tr>
</tbody>
</table>
**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).

**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.

![Diamond Sampling Rules Diagram](image)

The solid-colored subpixels are considered *interior* to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive**. This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range (1, +infinity] or [-infinity, -1) touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) – where if the right corner was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).

3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this *passing through* the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as *inside* for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```plaintext
delta_x = point.x - sample.x
delta_y = point.y - sample.y
distance = abs(delta_x) + abs(delta_y)
interior = (distance < 0.5)
bottom_corner = (delta_x == 0.0) && (delta_y == 0.5)
left_corner = (delta_x == -0.5) && (delta_y == 0.0)
right_corner = (delta_x == 0.5) && (delta_y == 0.0)
bottom_left_edge = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)
inside = interior || bottom_corner || (slopePosOne ? left_corner : right_corner) || (slopePosOne && left_edge) || (slopeNegOne && right_edge)
```

**GIQ (Diamond) Sampling Rules – DX10 Mode**

When the **Legacy Line Rasterization Enable** bit in WM_STATE is **DISABLED**, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_**xxx** primitive or the last pixel of each line in a LINELIST_**xxx** primitive is rendered.
Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.

The solid-colored subpixels are considered **interior** to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.

2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as -1 ≤ slope ≤ 1).** Including the right corner subpixel ensures that lines with slopes in the range (>1, +infinity] or [-infinity, <-1) touch a diamond even when they cross exactly between pixel diamonds.

3. **The left corner subpixel is never inclusive.** For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).

4. **The subpixels along the bottom left edge are always inclusive.** This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this **passing through** the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be
lit, this case would cause the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as inside, 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.

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 \((\text{point}.x, \text{point}.y)\) is inside the diamond of the pixel sample point \((\text{sample}.x, \text{sample}.y)\), given additional information about the slope (XMajor).

\[
\begin{align*}
\text{delta}_x & \quad = \text{point}.x - \text{sample}.x \\
\text{delta}_y & \quad = \text{point}.y - \text{sample}.y \\
\text{distance} & \quad = \text{abs} (\text{delta}_x) + \text{abs} (\text{delta}_y) \\
\text{interior} & \quad = (\text{distance} < 0.5) \\
\text{bottom}_\text{corner} & \quad = (\text{delta}_x == 0.0) \&\& (\text{delta}_y == 0.5) \\
\text{left}_\text{corner} & \quad = (\text{delta}_x == -0.5) \&\& (\text{delta}_y == 0.0) \\
\text{right}_\text{corner} & \quad = (\text{delta}_x == 0.5) \&\& (\text{delta}_y == 0.0) \\
\text{bottom}_\text{left}_\text{edge} & \quad = (\text{distance} == 0.5) \&\& (\text{delta}_x < 0) \&\& (\text{delta}_y > 0) \\
\text{bottom}_\text{right}_\text{edge} & \quad = (\text{distance} == 0.5) \&\& (\text{delta}_x > 0) \&\& (\text{delta}_y > 0) \\
\text{inside} & \quad = \text{interior} || \text{bottom}_\text{corner} || (!\text{XMajor} \&\& \text{right}_\text{corner}) || (\text{bottom}_\text{left}_\text{edge}) || (\text{bottom}_\text{right}_\text{edge})
\end{align*}
\]

**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 (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 \( \text{LineAARegion}/2 > \text{LineWidth}/2 \), 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.

### 3DSTATE_SF

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 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.

SF_CLIP_VIEWPORT

SCISSOR_RECT
Attribute Interpolation Setup

With the attribute interpolation setup function being implemented in hardware, a number of state fields in 3DSTATE_SF are utilized to control interpolation setup.

**Number of SF Output Attributes** sets the number of attributes that will be output from the SF stage, not including position. This can be used to specify up to 32, and may differ from the number of input attributes. The number of input attributes is derived from the **Vertex URB Entry Read Length** field. Note that this field is also used to specify whether swizzling is to be performed on Attributes 0-15 or Attributes 16-32. See the state field definition for details.

Attribute Swizzling

The first or last set of 16 attributes can be swizzled according to certain state fields. **Attribute Swizzle Enable** enables the swizzling for all 16 of these attributes, and each of the attributes has a 2-bit **Swizzle Select** field that controls swizzling with the following settings:

- **INPUTATTR** – This attribute is sourced from AttrInputReg[SourceAttribute].
- **INPUTATTR_FACING** – This attribute is sourced from AttrInputReg[SourceAttribute] if the object is front-facing, otherwise it is sourced from AttrInputReg[SourceAttribute+1].
- **INPUTATTR_W** – This attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination).
- **INPUTATTR_FACING** – If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination). If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. WYZW.

Each of the first or last set of 16 attributes also has a 5-bit **Source Attribute** field which specify, per output attribute (not component), which input attribute sources the output attribute when INPUTATTR is selected for **Swizzle Select**. A **Source Attribute** value of 0 corresponds to the 128-bit attribute immediately following the vertex 4D position. If INPUTATTR_FACING is selected, this specifies the first of two consecutive (front, back) input attributes, where the SourceAttribute value can be an odd or even number (just not 31, as that would place the back-face input attribute past the end of the input max complement of input attributes).

Constant overriding is also available for the first or last set of 16 attributes. Each attribute has a **Constant Source** field which specifies the constant values per swizzled attribute, with the following settings available:

- **XYZW = 0000**
- **XYZW = 0001**
- **XYZW = 1111**

Each channel of each attribute has a **Component Override** field to control whether the corresponding channel is overridden with the constant value defined in **Constant Source**.
Interpolation Modes

All 32 attributes have a **Constant Interpolation Enable** state field bit to specify whether all components of the *post-swizzled* attribute are to be interpolated as constant values (not varying over the pixels of the object). If set, the attribute at the provoking vertex is copied to a0, and a1 and a2 are set to zero – this results in a constant interpolation of the provoking vertex value. If clear, the attribute is linearly interpolated. Attributes 0-15 are further subjected to Wrap Shortest processing on a per-component basis, via the **Attribute WrapShortest Enables** state bitfields. WrapShortest processing modifies the a1 and/or a2 values depending on attribute deltas. All

The table below indicates the output values of a0, a1, and a2 depending on interpolation mode settings.

<table>
<thead>
<tr>
<th>Mode</th>
<th>a0</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>A0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Linear</td>
<td>A0</td>
<td>A1-A0</td>
<td>A2-A0</td>
</tr>
<tr>
<td>Wrap Shortest</td>
<td>(A1-A0)+1</td>
<td>(A1-A0) &lt;= -0.5</td>
<td>(A2-A0)+1</td>
</tr>
<tr>
<td></td>
<td>(A1-A0)-1</td>
<td>(A1-A0) &gt;= 0.5</td>
<td>(A2-A0)-1</td>
</tr>
<tr>
<td></td>
<td>(A1-A0)</td>
<td>otherwise</td>
<td>(A2-A0)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERLEFT</td>
<td>(0,0,0,1)</td>
<td>(1,0,0,1)</td>
</tr>
<tr>
<td>Top</td>
<td>(0,1,0,1)</td>
<td>(1,1,0,1)</td>
</tr>
<tr>
<td>Bottom</td>
<td>(0,0,1)</td>
<td>(1,0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Origin</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWERLEFT</td>
<td>(0,1,0,1)</td>
<td>(1,1,0,1)</td>
</tr>
<tr>
<td>Top</td>
<td>(0,1,0,1)</td>
<td>(1,1,0,1)</td>
</tr>
<tr>
<td>Bottom</td>
<td>(0,0,0,1)</td>
<td>(1,0,0,1)</td>
</tr>
</tbody>
</table>

The state used by "setup backend" is defined by the following inline state packet.
**3DSTATE_SBE**

**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 \((a_0, a_1, \text{ and } a_2)\) 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 \(a_0/a_1/a_2\) and the pixel/sample’s \(b_1/b_2\) barycentric parameters, where \(A\) is the value of the attribute channel at that pixel/sample:

\[
A = a_0 + (a_1 \times b_1) + (a_2 \times b_2)
\]
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

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 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

**Programming Note:** WM Unit also receives 3DSTATE_WM_HZ_OP, 3DSTATE_RASTER, 3DSTATE_MULTISAMPLE, 3DSTATE_WM_CHROMAKEY, 3DSTATE_PS_BLEND, and 3DSTATE_PS_EXTRA.

**3DSTATE_SAMPLE_MASK**

The following inline state packets define the sample mask state used by the windower stage for different generations.

3DSTATE_SAMPLE_MASK

<table>
<thead>
<tr>
<th>State</th>
<th>Stencil buffer Clear</th>
<th>Depth buffer clear</th>
<th>Depth Buffer Resolve Enable</th>
<th>Hierarchical Depth Buffer Resolve Enable</th>
<th>Project</th>
</tr>
</thead>
</table>

**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

<table>
<thead>
<tr>
<th>Pixel 0</th>
<th>Pixel 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel 2</td>
<td>Pixel 3</td>
</tr>
</tbody>
</table>

Drawing Rectangle Clipping

The Drawing Rectangle defines the maximum extent of pixels which can be rendered. Portions of objects falling outside the Drawing Rectangle will be clipped (pixels discarded). Implementations will typically discard objects falling completely outside of the Drawing Rectangle as early in the pipeline as possible. There is no control to turn off Drawing Rectangle clipping – it is unconditional.

For the purposes of clipping, the Drawing Rectangle must itself be clipped to the destination buffer extents (The Drawing Rectangle Origin, used to offset relative X,Y coordinates earlier in the pipeline, is permitted to lie offscreen). The **Clipped Drawing Rectangle X,Y Min,Max** state variables (programmed via 3DSTATE_DRAWING_RECTANGLE – See SF Unit) defines the intersection of the Drawing Rectangle and the Color Buffer. It is specified with non-negative integer pixel coordinates relative to the Destination Buffer upper-left origin.

Pixels with coordinates outside of the Drawing Rectangle cannot be rendered (i.e., the rectangle is inclusive). For example, to render to a full-screen 1280x1024 buffer, the following values would be required: Xmin=0, Ymin=0, Xmax=1279 and Ymax=1023.

For full screen rendering, the Drawing Rectangle coincides with the screen-sized buffer. For front-buffer windowed rendering it coincides with the destination window.

Line Rasterization

See SF Unit chapter for details on the screen-space geometry of the various line types.

Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

3DSTATE_AA_LINE_PARAMS

3DSTATE_AA_LINE_PARAMETERS
The slope and bias values should be computed to closely match the reference rasterizer results. Based on empirical data, the following recommendations are offered:

The final alpha for the center of the line needs to be 148 to match the reference rasterizer. In this case, the Lo to edge 0 and edge 3 will be the same. Since the alpha for each edge is multiplied together, we get:

\[ \text{edge0alpha} \times \text{edge1alpha} = 148/255 = 0.580392157 \]

Since \( \text{edge0alpha} = \text{edge3alpha} \) we get:

\[ (\text{edge0alpha})^2 = 0.580392157 \]

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

The desired alpha for pixel 1 = 54/255 = 0.211764706

The slope is \( (0.761834731 – 0.211764706) = 0.550070025 \)

Since we are using 8 bit precision, the slope becomes

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

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:

\[ b = \lfloor s/r \rfloor \mod 16. \]

A pixel is rendered if the \( b \)th bit of \( p \) is 1, otherwise it is discarded. The bits of \( p \) are numbered with 0 being the least significant and 15 being the most significant.

**3DSTATE_LINE_STIPPLE**
Polygon (Triangle and Rectangle) Rasterization

The rasterization of LINE, TRIANGLE, and RECTANGLE objects into pixels requires a pixel sampling grid to be defined. This grid is defined as an axis-aligned array of pixel sample points spaced exactly 1 pixel unit apart. If a sample point falls within one of these objects, the pixel associated with the sample point is considered inside the object, and information for that pixel is generated and passed down the pipeline.

For TRIANGLE and RECTANGLE objects, if a sample point intersects an edge of the object, the associated pixel is considered inside the object if the intersecting edge is a left or top edge (or, more exactly, the intersected edge is not a right or bottom edge). Note that top and bottom edges are by definition exactly horizontal. See TRIANGLE and RECTANGLE Edge Types below for the edge types for representative TRIANGLE and RECTANGLE objects (solid edges are inclusive, dashed edges are exclusive).

TRIANGLE and RECTANGLE Edge Types

Polygon Stipple

The Polygon Stipple function, controlled via the Polygon Stipple Enable state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered. Polygon stipple is applied only to the following primitive types:

| 3DPRIM_POLYGON |
| 3DPRIM_TRIFAN |
| 3DPRIM_TRILIST |
| 3DPRIM_TRISTRIP |
| 3DPRIM_TRISTRIP_REVERSE |
Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command. The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

**3DSTATE_POLY_STIPPLE_OFFSET**  
**3DSTATE_POLY_STIPPLE_PATTERN**  

**Multisampling**

The multisampling function has two components:

- **Multisample Rasterization**: multisample rasterization occurs at a subpixel level, wherein each pixel consists of a number of “samples” at state-defined positions within the pixel footprint. Coverage of the primitive as well as color calculator operations (stencil test, depth test, color buffer blending, etc.) are done at the sample level. In addition, the pixel shader itself can optionally run at the sample level depending on a separate state field.

- **Multisample Render Targets (MSRT)**: The render targets, as well as the depth and stencil buffers, now have the ability to store per-sample values. When combined with multisample rasterization, color calculator operations such as stencil test, depth test, and color buffer blending are done with the destination surface containing potentially different values per sample.

**3DSTATE_MULTISAMPLE**  
**3DSTATE_RAST_MULTISAMPLE**  

**Multisample Modes State**

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.

<table>
<thead>
<tr>
<th>State Element</th>
<th>Project</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multisample Rasterization Mode</td>
<td></td>
<td>3DSTATE_SF and 3DSTATE_WM</td>
<td>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). The mode is controlled directly.</td>
</tr>
<tr>
<td>Multisample Dispatch</td>
<td></td>
<td>3DSTATE_WM</td>
<td>Controls whether the pixel shader is executed per pixel</td>
</tr>
<tr>
<td>State Element</td>
<td>Project</td>
<td>Source</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td></td>
<td>or per sample.</td>
</tr>
<tr>
<td>Number of Multisamples</td>
<td>3DSTATE_MULTISAMPLE and SURFACE_STATE</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Rast Number of Samples</td>
<td>3DSTATE_RAST_MULTISAMPLE::Number of Rasterization Multisamples</td>
<td>Indicates the number of samples per pixel using RTIR rather than MSAA.</td>
<td></td>
</tr>
<tr>
<td>RTIR Enabled</td>
<td>3DSTATE_SF::RT Independent Rasterization Enable == 1</td>
<td>Enable Render Target Independent Rasterization.</td>
<td></td>
</tr>
<tr>
<td>Pixel Location</td>
<td>3DSTATE_MULTISAMPLE</td>
<td>Indicates the subpixel location where values specified as &quot;pixel&quot; are sampled. This is either the upper left corner or the center.</td>
<td></td>
</tr>
<tr>
<td>MSAA Sample Offsets</td>
<td>3DSTATE_MULTISAMPLE</td>
<td>For each of the N samples, specifies the subpixel location of each sample.</td>
<td></td>
</tr>
<tr>
<td>RTIR Sample Offsets</td>
<td>3DSTATE_RAST_MULTISAMPLE</td>
<td>For each of the N samples, specifies the subpixel location of each sample.</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Project</th>
<th>AA Line Support Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>3DSTATE_SF::Anti-aliasing Enable == 1</td>
</tr>
</tbody>
</table>
- **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

<table>
<thead>
<tr>
<th>Multisample Rasterization Mode</th>
<th>Anti-Aliasing Enable</th>
<th>SF_STATE::Line Width</th>
<th>Line Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF_*</td>
<td>0</td>
<td>Non-Zero</td>
<td>Non-Anti-aliased Wide Lines</td>
</tr>
<tr>
<td>OFF_*</td>
<td>0</td>
<td>0.0</td>
<td>Diamond Lines</td>
</tr>
<tr>
<td>OFF_*</td>
<td>1</td>
<td>Non-Zero</td>
<td>See Note A below.</td>
</tr>
<tr>
<td>OFF_*</td>
<td>1</td>
<td>0.0</td>
<td>Diamond Lines</td>
</tr>
<tr>
<td>ON_*</td>
<td>*</td>
<td>*</td>
<td>MSAA Lines</td>
</tr>
</tbody>
</table>

**Note A: Anti-Aliasing Details for Table 2**

<table>
<thead>
<tr>
<th>Project</th>
<th>Anti-Aliasing Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>Anti-Aliased Lines with Alpha Coverage</td>
</tr>
</tbody>
</table>

### Table 3: Multisample Modes with RTIR Disabled

<table>
<thead>
<tr>
<th>Number of Multisamples</th>
<th>MS RAST MODE</th>
<th>MS DISP MODE</th>
<th>HW Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMSAMPLES_1</td>
<td>OFF_PIXEL</td>
<td>PERSAMPLE</td>
<td><strong>Legacy Non-MSAA Mode</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X rasterization, using Pixel Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Legacy lines or AA-line rasterization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X PS, sample at Pixel Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X output merge, eval Depth at Pixel Location</td>
</tr>
<tr>
<td>ON_PIXEL</td>
<td>PERSAMPLE</td>
<td></td>
<td><strong>1X Multisampling Mode</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X rasterization, using Pixel Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSAA lines only, using Pixel Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X PS, sample at Pixel Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X output merge, eval Depth at Pixel Location</td>
</tr>
<tr>
<td>-</td>
<td>PERPIXEL</td>
<td></td>
<td>Treated the same as PERSAMPLE</td>
</tr>
<tr>
<td>ON_PATTERN</td>
<td>-</td>
<td></td>
<td>Invalid</td>
</tr>
<tr>
<td>Project:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFF_PATTERN</td>
<td>-</td>
<td>Invalid</td>
<td></td>
</tr>
</tbody>
</table>
| 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 |
### Project:

<table>
<thead>
<tr>
<th>Project</th>
<th>Line Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>PERSAMPLE Mixed Mode, PerSample PS</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Line Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>ON_PATTERN Pattern MSAA, PerPixel PS</td>
</tr>
<tr>
<td></td>
<td>nX rasterization, using Sample Offsets MSAA lines only 1X PS, sample at Pixel Location nX output merge, eval depth at Sample Offsets</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Line Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>PERSAMPLE Pattern MSAA, PerSample PS</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
</tbody>
</table>

**Note B: Line Details for Table 3 and Table 4**

<table>
<thead>
<tr>
<th>Project</th>
<th>Line Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Table 4: Multisample Modes with RTIR Enabled**

<table>
<thead>
<tr>
<th>Rast Number of Samples</th>
<th>MS RAST MODE</th>
<th>HW Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMRASTSAMPLES_1</td>
<td>OFF_PIXEL</td>
<td>Legacy Non-MSAA Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1X rasterization, using Pixel Location</td>
</tr>
</tbody>
</table>
### Project:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ON_PIXEL</td>
<td><strong>1X Multisampling Mode</strong></td>
</tr>
<tr>
<td></td>
<td>1X rasterization, using Pixel Location</td>
</tr>
<tr>
<td></td>
<td>MSAA lines only, using Pixel Location</td>
</tr>
<tr>
<td></td>
<td>1X PS, sample at Pixel Location</td>
</tr>
<tr>
<td></td>
<td>1X output merge, eval Depth at Pixel Location</td>
</tr>
<tr>
<td>ON_PATTERN</td>
<td>Invalid</td>
</tr>
<tr>
<td>OFF_PATTERN</td>
<td>Invalid</td>
</tr>
<tr>
<td>OFF_PIXEL</td>
<td>Invalid</td>
</tr>
<tr>
<td>ON_PIXEL</td>
<td>Invalid</td>
</tr>
<tr>
<td>OFF_PATTERN</td>
<td><strong>Mixed Mode, PerPixel PS</strong></td>
</tr>
<tr>
<td></td>
<td>See Note B above.</td>
</tr>
<tr>
<td></td>
<td>Non-Lines: nX rasterization, using Sample Offsets</td>
</tr>
<tr>
<td></td>
<td>1X PS, sample at Pixel Location</td>
</tr>
<tr>
<td></td>
<td>1X output merge, eval depth at Pixel Location</td>
</tr>
<tr>
<td>ON_PATTERN</td>
<td><strong>Pattern RTIR, PerPixel PS</strong></td>
</tr>
<tr>
<td></td>
<td>nX rasterization, using Sample Offsets</td>
</tr>
<tr>
<td></td>
<td>MSAA lines only</td>
</tr>
<tr>
<td></td>
<td>1X PS, sample at Pixel Location</td>
</tr>
<tr>
<td></td>
<td>1X output merge, eval Depth at Pixel Location</td>
</tr>
</tbody>
</table>

**Note:** Multisample Dispatch Mode is not taken into account in Table 4 given that with RTIR:

<table>
<thead>
<tr>
<th>Project</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>The value of PERSAMPLE for this state variable is invalid.</td>
</tr>
</tbody>
</table>

### 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.

Other WM Functions

The only other WM function is Statistics Gathering.

Statistics Gathering

If Statistics Enable is set in WM_STATE or 3DSTATE_WM, the Windower increments the PS_INVOCATIONS_COUNT register once for each unmasked pixel (or sample) that is dispatched to a Pixel Shader thread.

If Early Depth Test Enable is set it is possible for pixels or samples to be discarded before reaching the Pixel Shader due to failing the depth or stencil test. PS_INVOCATIONS_COUNT will still be incremented for these pixels or samples since the depth test occurs after the pixel shader from the point of view of SW.
Pixel

This section contains the following subsections:

- **Depth and Stencil**, which covers the Depth and Stencil test functions
- **Pixel Dispatch**, which covers pixel shader state, pixel grouping, multisampling effects on pixel shader dispatch, and pixel shader thread payload
- **Pixel Backend**, which covers backend processing
Early Depth/Stencil Processing

The Windower/IZ unit provides the Early Depth Test function, a major performance-optimization feature where an attempt is made to remove pixels that fail the Depth and Stencil Tests prior to pixel shading. This requires the WM unit to perform the interpolation of pixel (source) depth values, read the current (destination) depth values from the cached depth buffer, and perform the Depth and Stencil Tests. As the WM unit has per-pixel source and destination Z values, these values are passed in the PS thread payload, if required.

Depth Offset

**Note:** The depth offset function is contained in SF unit, thus the state to control it is also contained in SF unit.

There are occasions where the Z position of some objects need to be slightly offset to reduce artifacts due to coplanar or near-coplanar primitives. A typical example is drawing the edges of triangles as wireframes—the lines need to be drawn slightly closer to the viewer to ensure they will not be occluded by the underlying polygon. Another example is drawing objects on a wall—without a bias on the z positions, they might be fully or partially occluded by the wall.

The device supports *global* depth offset, applied only to triangles, that bases the offset on the object’s z slope. Note that there is no clamping applied at this stage after the Z position is offset—clamping to [0,1] can be performed later after the Z position is interpolated to the pixel. This is preferable to clamping prior to interpolation, as the clamping would change the Z slope of the entire object.

The Global Depth Offset function is controlled by the **Global Depth Offset Enable** state variable in WM_STATE. Global Depth Offset is only applied to 3DOBJ_TRIANGLE objects.

When Global Depth Offset Enable is ENABLED, the pipeline will compute:

MaxDepthSlope = max(abs(dZ/dX),abs(dz/dy)) // approximation of max depth slope for polygon

When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

\[
\text{Bias} = \text{GlobalDepthOffsetConstant} \times r + \text{GlobalDepthOffsetScale} \times \text{MaxDepthSlope}
\]

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:

\[
\text{Bias} = \text{GlobalDepthOffsetConstant} \times 2^{\text{exponent}\text{(max z in primitive)} - r} + \text{GlobalDepthOffsetScale} \times \text{MaxDepthSlope}
\]

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 GobalDepthOffsetConstant).
Adding Bias to z:

```plaintext
if (GlobalDepthOffsetClamp > 0)
    Bias = min(DepthBiasClamp, Bias)
elif (GlobalDepthOffsetClamp < 0)
    Bias = max(DepthBiasClamp, Bias)
// else if GlobalDepthOffsetClamp == 0, no clamping occurs
z = z + Bias
```

Biasing is constant for a given primitive. The biasing formulas are performed with float32 arithmetic. Global Depth Bias is not applied to any point or line primitives.

**Early Depth Test/Stencil Test/Write**

When **Early Depth Test Enable** is ENABLED, the WM unit will attempt to discard depth-occluded pixels during scan conversion (before processing them in the Pixel Shader). Pixels are only discarded when the WM unit can ensure that they would have no impact to the ColorBuffer or DepthBuffer. This function is therefore only a performance feature.

**Note:** **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.

<table>
<thead>
<tr>
<th>State Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pixel Shader Kill Pixel</strong></td>
<td>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 <code>killpixel</code> or <code>alphatest</code> function or as dictated by the results of the sampling of a <code>chroma-keyed</code> 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.</td>
</tr>
<tr>
<td><strong>Pixel Shader Computed Depth</strong></td>
<td>This must be set when the PS thread computes the source depth value (i.e., from the API POV, writes to the <code>oDepth</code> 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.</td>
</tr>
<tr>
<td><strong>Pixel Shader Uses Source</strong></td>
<td>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</td>
</tr>
</tbody>
</table>
### 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 Z is the 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.

#### Project:

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.

#### Project:

Since Hierarchical Depth Buffer supports multiple LODs. The HZ_height is different as shown in the table below:

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>HZ_Width (Bytes)</th>
<th>HZ_Height (Rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURFTYPE_1D</td>
<td>ceiling(Z_Width / 16) * 16</td>
<td>Ceiling (( Q_pitch * Z_depth/2) /8 ) * 8</td>
</tr>
<tr>
<td>SURFTYPE_2D</td>
<td>ceiling(Z_Width / 16) * 16</td>
<td>Ceiling (( Q_pitch * Z_depth/2) /8 ) * 8</td>
</tr>
<tr>
<td>SURFTYPE_3D</td>
<td>ceiling(Z_Width / 16) * 16</td>
<td>see below</td>
</tr>
<tr>
<td>SURFTYPE_CUBE</td>
<td>ceiling(Z_Width / 16) * 16</td>
<td>Ceiling (( Q_pitch * Z_depth * 6/2) /8 ) * 8</td>
</tr>
</tbody>
</table>

Where, Qpitch is computed using vertical alignment j=8. Please refer to the GPU overview volume for Qpitch definition.

The minimum HZ_Height required for a 3D surface must be computed based on hL parameters documented in the GPU Overview volume, and the maximum LOD m:

\[
HZ\_Height = \frac{1}{2} \left[ \sum_{i=0}^{m} h_i \cdot \max\left(1, \floor\left(\frac{Z\_Depth}{2^i}\right)\right) \right]
\]
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:

\[
HZ_{\text{QPitch}} = h_0 + \max \left( h_1, \sum_{i=2}^{m} h_i \right)
\]

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 \( m \):

\[
HZ_{\text{Height}} = \frac{1}{2} \left[ \sum_{i=0}^{m} h_i \times \left\lfloor \frac{Z_{\text{Depth}}}{2^i} \right\rfloor \right]
\]

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 NaN values are also allowed.
• The following alignment restrictions need to be met while doing the fast-clears.

<table>
<thead>
<tr>
<th>Project</th>
<th>Alignment Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>If <code>Number of Multisamples</code> 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.</td>
</tr>
<tr>
<td>HSW</td>
<td>If <code>Number of Multisamples</code> 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.</td>
</tr>
<tr>
<td>HSW</td>
<td>If <code>Number of Multisamples</code> 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.</td>
</tr>
<tr>
<td>HSW</td>
<td>If <code>Number of Multisamples</code> 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.</td>
</tr>
</tbody>
</table>

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 DEPTHSTALL bit and Depth FLUSH bits "set" before starting to render. DepthStall and DepthFlush are not needed between consecutive depth clear passes nor is it required if the depth-clear pass was done with "full_surf_clear" bit set in the 3DSTATE_WM_HZ_OP.

**Note:** If using the optimized depth buffer clear, this pipecontrol should be done after the resetting of the clear/resolve bits in the 3DSTATE_WM_HZ_OP (step #8).

### Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete. If the depth buffer is retained and used for another purpose (i.e. as input to the sampling engine as a shadow map), it must first be "resolved". This is done by setting the **Depth Buffer Resolve Enable** field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation.

Depth buffer can be used as a source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

The following is required when performing a depth buffer resolve:

- 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.

### 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.

**Separate Stencil Buffer**

The following tables describe the separate stencil buffer for different generations.

<table>
<thead>
<tr>
<th>Project:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
<tr>
<td>The stencil buffer has a format of R8_UNIT, and shares <strong>Surface Type</strong>, <strong>Height</strong>, <strong>Width</strong>, and <strong>Depth</strong>, <strong>Minimum Array Element</strong>, <strong>Render Target View Extent</strong>, <strong>Depth Coordinate Offset X/Y</strong>, <strong>LOD</strong>, and <strong>Depth Buffer Object Control State</strong> fields of the depth buffer.</td>
</tr>
</tbody>
</table>

**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, UAves 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
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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>E cannot be used in PERSAMPLE mode with number of multisamples &gt;= 2.</td>
<td></td>
</tr>
</tbody>
</table>

### Variable Pixel Dispatch

<table>
<thead>
<tr>
<th>Contiguous 64 Pixel Dispatch Enable</th>
<th>Contiguous 32 Pixel Dispatch Enable</th>
<th>32 Pixel Dispatch Enable</th>
<th>16 Pixel Dispatch Enable</th>
<th>8 Pixel Dispatch Enable</th>
<th>Valid</th>
<th>IP for n-pixel Contiguous Dispatch n=64</th>
<th>IP for n-pixel Contiguous Dispatch n=32</th>
<th>IP for n-pixel Contiguous Dispatch n=16</th>
<th>IP for n-pixel Contiguous Dispatch n=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>KSP[2]</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>E</td>
<td>KSP[2]</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>KSP[2]</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>KSP[2]</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>KSP[2]</td>
<td>KSP[2]</td>
<td>KSP[0]</td>
<td>KSP[0]</td>
</tr>
</tbody>
</table>

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.

```c
if ( 32PixelDispatchEnable && n > 7 )
    Dispatch 32 Pixels
else if ( 16PixelDispatchEnable && ( n > 2 || ! 8PixelDispatchEnable) )
    Dispatch 16 Pixels
else
    Dispatch 8 Pixels
end if
```

**Contiguous Dispatch Modes**

There are three cases to consider depending on which dispatch modes are enabled based on the legal combinations in the table above:

- **Only normal dispatch modes are enabled.** This is the normal operating mode in which all features are supported.

- **Only contiguous dispatch modes are enabled.** In this case, software must ensure that the fast composite restrictions are met.

- **Both normal and contiguous dispatch modes are enabled.** In this case, a combination of software and the setup kernel must check all of the restrictions required by the contiguous dispatch pixel shader code. The result of the check in the setup kernel is indicated in the message descriptor of the URB write message. The windower then chooses a dispatch mode from either the normal category or the contiguous category depending on whether the restriction check fails or passes, respectively.

If both the 32- and 64-pixel contiguous dispatch modes are enabled together, the windower chooses which one to use based on whether at least one pixel from the upper and lower 8x4 halves of the 8x8 block is active. If one half has no pixel active, the half that does have pixels active is dispatched as a 32-pixel thread.
The following logic describes how the windower chooses the dispatch mode based on which modes are enabled:

\[
d32 = \text{normal 32-pixel dispatch mode enabled}
\]
\[
d16 = \text{normal 16-pixel dispatch mode enabled}
\]
\[
d8 = \text{normal 8-pixel dispatch mode enabled}
\]
\[
c64 = \text{contiguous 64-pixel dispatch mode enabled}
\]
\[
c32 = \text{contiguous 32-pixel dispatch mode enabled}
\]

\[
\text{ContiguousSelect} = (c64 \lor c32) \land \neg (d32 \lor d16 \lor d8) \land \text{RestrictionCheckPass}
\]

### Table: For ContiguousSelect true:

<table>
<thead>
<tr>
<th>contiguous area available</th>
<th>first priority</th>
<th>second priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>both superspan halves</td>
<td>c64</td>
<td>c32</td>
</tr>
<tr>
<td>one superspan half</td>
<td>c32</td>
<td>c64</td>
</tr>
</tbody>
</table>

### Table: For ContiguousSelect false:

<table>
<thead>
<tr>
<th>subspans available</th>
<th>first priority</th>
<th>second priority</th>
<th>third priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s \geq 4)</td>
<td>d32</td>
<td>d16</td>
<td>d8</td>
</tr>
<tr>
<td>(4 &gt; s \geq 2)</td>
<td>d16</td>
<td>d8</td>
<td>d32</td>
</tr>
<tr>
<td>(2 &gt; s \geq 1)</td>
<td>d8</td>
<td>d16</td>
<td>d32</td>
</tr>
</tbody>
</table>

### 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 be required to process all samples for a single subspan. In the latter case, the `StartingSamplePairIndex` value is included in the payload header so the Render Target Write message will access the correct samples with each message.
**PERSAMPLE SIMD8 4X Dispatch**

The following table provides the complete dispatch/slot mappings for all the MS/Dispatch combinations.

<table>
<thead>
<tr>
<th>Dispatch Size</th>
<th>Num Samples</th>
<th>Slot Mapping (SSPI = Starting Sample Pair Index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD32</td>
<td>1X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>4X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>8X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td>Dispatch Size</td>
<td>Num Samples</td>
<td>Slot Mapping</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SSPI = Starting Sample Pair Index)</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>4X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>8X</td>
<td>Dispatch[i]; (i=0, 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSPI = i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]</td>
</tr>
<tr>
<td>SIMD8</td>
<td>1X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td>Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]</td>
</tr>
<tr>
<td></td>
<td>4X</td>
<td>Dispatch[i]; (i=0..1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSPI = i</td>
</tr>
<tr>
<td>Dispatch Size</td>
<td>Num Samples</td>
<td>Slot Mapping (SSPI = Starting Sample Pair Index)</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-----------------------------------------------</td>
</tr>
</tbody>
</table>
| 8X            |             | 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 (UNRESOLVED CROSS REFERENCE, PS Thread Payload for Normal Dispatch) applies to . 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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30:24</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
|       | 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.  
<p>|       |      | Format: Reserved for HW Implementation Use. |         |
| R0.5  | 31:10| <strong>Scratch Space Pointer:</strong> 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. |         |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9:8</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td>R0.4</td>
<td>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]</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>R0.3</td>
<td>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]</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>R0.2</td>
<td>Reserved: Delivered as zeros (reserved for message header fields).</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>R0.1</td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>5:0</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>R0.0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>30:27</td>
<td>Viewport Index: Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26:16</td>
<td>Render Target Array Index: Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index Range = [0,2047] SURFTYPE_2D: specifies the array index Range = [0,2047] SURFTYPE_3D: specifies the &quot;r&quot; coordinate Range = [0,2047]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SURFTYPE_CUBE: specifies the face identifier Range = [0, 5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Face Index</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+x 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-x 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+y 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-y 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+z 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-z 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U11</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td><strong>Front/Back Facing Polygon:</strong> Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Front Facing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Back Facing</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td><strong>Source Depth to Render Target:</strong> Indicates that source depth will be sent to the render target.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td><strong>oMask to Render Target:</strong> Indicates that oMask will be sent to the render target.</td>
<td></td>
</tr>
<tr>
<td>11:9</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Reserved for expansion of <strong>Starting Sample Pair Index.</strong></td>
<td></td>
</tr>
<tr>
<td>7:6</td>
<td></td>
<td><strong>Starting Sample Pair Index:</strong> Indicates the index of the first sample pair of the dispatch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0, 3]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>4:0</td>
<td></td>
<td><strong>Primitive Topology Type:</strong> This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST). Format: (See 3DPRIMITIVE command in 3D Pipeline.)</td>
<td></td>
</tr>
<tr>
<td>R1.7</td>
<td>31:16</td>
<td><strong>Pixel/Sample Mask (SubSpan[3:0]):</strong> 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. <strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Pixel/Sample Mask Copy (SubSpan[3:0]):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>R1.6</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>R1.5</td>
<td>31:16</td>
<td><strong>Y3:</strong> Y coordinate (screen space) for upper-left pixel of subspan 3 (slot 12). <strong>X3:</strong> X coordinate (screen space) for upper-left pixel of subspan 3 (slot 12). Format = U16</td>
<td></td>
</tr>
<tr>
<td>R1.4</td>
<td>31:16</td>
<td><strong>Y2:</strong> Y coordinate (screen space) for upper-left pixel of subspan 2 (slot 8). <strong>X2:</strong> X coordinate (screen space) for upper-left pixel of subspan 2 (slot 8). Format = U16</td>
<td></td>
</tr>
<tr>
<td>R1.3</td>
<td>31:16</td>
<td><strong>Y1:</strong> Y coordinate (screen space) for upper-left pixel of subspan 1 (slot 4). <strong>X1:</strong> X coordinate (screen space) for upper-left pixel of subspan 1 (slot 4). Format = U16</td>
<td></td>
</tr>
<tr>
<td>R1.2</td>
<td>31:16</td>
<td><strong>Y0:</strong> Y coordinate (screen space) for upper-left pixel of subspan 0 (slot 0). <strong>X0:</strong> X coordinate (screen space) for upper-left pixel of subspan 0 (slot 0). Format = U16</td>
<td></td>
</tr>
<tr>
<td>R1.1</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>R1.0</td>
<td>31:20</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Slot 3 SampleID (if pixel or sample dispatch)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1X MSAA range: [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2X MSAA range [0,1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4X MSAA range [0..3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8X MSAA range [0..7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Slot 2 SampleID (if pixel or sample dispatch)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1X MSAA range: [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2X MSAA range [0,1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4X MSAA range [0..3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8X MSAA range [0..7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Slot 1 SampleID (if pixel or sample dispatch)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1X MSAA range: [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2X MSAA range [0,1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4X MSAA range [0..3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8X MSAA range [0..7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Slot 0 SampleID (if pixel or sample dispatch)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1X MSAA range: [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2X MSAA range [0,1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4X MSAA range [0..3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8X MSAA range [0..7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2:</td>
<td>Delivered only if this is a 32-pixel dispatch.</td>
<td></td>
</tr>
<tr>
<td>R2.7</td>
<td>31:16</td>
<td><strong>Pixel/Sample Mask (SubSpan[7:4]):</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>Pixel/Sample Mask Copy (SubSpan[7:4])</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>R2.6</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>R2.5</td>
<td>31:16</td>
<td><strong>Y7</strong>: Y coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>X7</strong>: X coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16</td>
<td></td>
</tr>
<tr>
<td>R2.4</td>
<td>31:16</td>
<td>Y6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>X6</td>
<td></td>
</tr>
<tr>
<td>R2.3</td>
<td>31:16</td>
<td>Y5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>X5</td>
<td></td>
</tr>
<tr>
<td>R2.2</td>
<td>31:16</td>
<td>Y4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>X4</td>
<td></td>
</tr>
<tr>
<td>R2.1</td>
<td>31:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>R2.0</td>
<td>31:16</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>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]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>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]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Slot 5 SampleID Format = U4</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>1X MSAA range: [0]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>2X MSAA range [0,1]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>4X MSAA range [0..3]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>8X MSAA range [0..7]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>16X MSAA range [0..15]</strong></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td><strong>Slot 4 SampleID</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>1X MSAA range: [0]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>2X MSAA range [0,1]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>4X MSAA range [0..3]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>8X MSAA range [0..7]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>16X MSAA range [0..15]</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R3-R26</strong>: Delivered only if the corresponding <strong>Barycentric Interpolation Mode</strong> bit is set. Register phases containing Slot 8-15 data are not delivered in 8-pixel dispatch mode.</td>
<td></td>
</tr>
<tr>
<td>R3.7</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 7</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase is only included if the corresponding enable bit in <strong>Barycentric Interpolation Mode</strong> is set.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td>R3.6</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 6</strong></td>
<td></td>
</tr>
<tr>
<td>R3.5</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 5</strong></td>
<td></td>
</tr>
<tr>
<td>R3.4</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 4</strong></td>
<td></td>
</tr>
<tr>
<td>R3.3</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 3</strong></td>
<td></td>
</tr>
<tr>
<td>R3.2</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 2</strong></td>
<td></td>
</tr>
<tr>
<td>R3.1</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 1</strong></td>
<td></td>
</tr>
<tr>
<td>R3.0</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 0</strong></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td><strong>Perspective Pixel Location Barycentric[2] for Slots 7:0</strong></td>
<td></td>
</tr>
<tr>
<td>R5.7</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 15</strong></td>
<td></td>
</tr>
<tr>
<td>R5.6</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 14</strong></td>
<td></td>
</tr>
<tr>
<td>R5.5</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 13</strong></td>
<td></td>
</tr>
<tr>
<td>R5.4</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 12</strong></td>
<td></td>
</tr>
<tr>
<td>R5.3</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 11</strong></td>
<td></td>
</tr>
<tr>
<td>R5.2</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 10</strong></td>
<td></td>
</tr>
<tr>
<td>R5.1</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 9</strong></td>
<td></td>
</tr>
<tr>
<td>R5.0</td>
<td>31:0</td>
<td><strong>Perspective Pixel Location Barycentric[1] for Slot 8</strong></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>Perspective Pixel Location Barycentric[2] for Slots 15:8</td>
<td></td>
</tr>
<tr>
<td>R7:10</td>
<td></td>
<td>Perspective Centroid Barycentric</td>
<td></td>
</tr>
<tr>
<td>R11:14</td>
<td></td>
<td>Perspective Sample Barycentric</td>
<td></td>
</tr>
<tr>
<td>R15:18</td>
<td></td>
<td>Linear Pixel Location Barycentric</td>
<td></td>
</tr>
<tr>
<td>R19:22</td>
<td></td>
<td>Linear Centroid Barycentric</td>
<td></td>
</tr>
<tr>
<td>R23:26</td>
<td></td>
<td>Linear Sample Barycentric</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R27</strong>: Delivered only if <strong>Pixel Shader Uses Source Depth</strong> is set.</td>
<td></td>
</tr>
<tr>
<td>R27.7</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase is only included if <strong>Pixel Shader Uses Source Depth</strong> (WM_STATE) is set.</td>
<td></td>
</tr>
<tr>
<td>R27.6</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 6</td>
<td></td>
</tr>
<tr>
<td>R27.5</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 5</td>
<td></td>
</tr>
<tr>
<td>R27.4</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 4</td>
<td></td>
</tr>
<tr>
<td>R27.3</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 3</td>
<td></td>
</tr>
<tr>
<td>R27.2</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 2</td>
<td></td>
</tr>
<tr>
<td>R27.1</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 1</td>
<td></td>
</tr>
<tr>
<td>R27.0</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R28</strong>: Delivered only if <strong>Pixel Shader Uses Source Depth</strong> is set and this is not an 8-pixel dispatch.</td>
<td></td>
</tr>
<tr>
<td>R28.7</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 15</td>
<td></td>
</tr>
<tr>
<td>R28.6</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 14</td>
<td></td>
</tr>
<tr>
<td>R28.5</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 13</td>
<td></td>
</tr>
<tr>
<td>R28.4</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 12</td>
<td></td>
</tr>
<tr>
<td>R28.3</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 11</td>
<td></td>
</tr>
<tr>
<td>R28.2</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 10</td>
<td></td>
</tr>
<tr>
<td>R28.1</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 9</td>
<td></td>
</tr>
<tr>
<td>R28.0</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R29</strong>: Delivered only if <strong>Pixel Shader Uses Source W</strong> is set.</td>
<td></td>
</tr>
<tr>
<td>R29.7</td>
<td>31:0</td>
<td>Interpolated W for Slot 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase are only included if <strong>Pixel Shader Uses Source W</strong> (WM_STATE) is set.</td>
<td></td>
</tr>
<tr>
<td>R29.6</td>
<td>31:0</td>
<td>Interpolated W for Slot 6</td>
<td></td>
</tr>
<tr>
<td>R29.5</td>
<td>31:0</td>
<td>Interpolated W for Slot 5</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>R29.4</td>
<td>31:0</td>
<td>Interpolated W for Slot 4</td>
<td></td>
</tr>
<tr>
<td>R29.3</td>
<td>31:0</td>
<td>Interpolated W for Slot 3</td>
<td></td>
</tr>
<tr>
<td>R29.2</td>
<td>31:0</td>
<td>Interpolated W for Slot 2</td>
<td></td>
</tr>
<tr>
<td>R29.1</td>
<td>31:0</td>
<td>Interpolated W for Slot 1</td>
<td></td>
</tr>
<tr>
<td>R29.0</td>
<td>31:0</td>
<td>Interpolated W for Slot 0</td>
<td></td>
</tr>
<tr>
<td><strong>R30:</strong> Delivered only if <strong>Pixel Shader Uses Source W</strong> is set and this is not an 8-pixel dispatch.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R30.7</td>
<td>31:0</td>
<td>Interpolated W for Slot 15</td>
<td></td>
</tr>
<tr>
<td>R30.6</td>
<td>31:0</td>
<td>Interpolated W for Slot 14</td>
<td></td>
</tr>
<tr>
<td>R30.5</td>
<td>31:0</td>
<td>Interpolated W for Slot 13</td>
<td></td>
</tr>
<tr>
<td>R30.4</td>
<td>31:0</td>
<td>Interpolated W for Slot 12</td>
<td></td>
</tr>
<tr>
<td>R30.3</td>
<td>31:0</td>
<td>Interpolated W for Slot 11</td>
<td></td>
</tr>
<tr>
<td>R30.2</td>
<td>31:0</td>
<td>Interpolated W for Slot 10</td>
<td></td>
</tr>
<tr>
<td>R30.1</td>
<td>31:0</td>
<td>Interpolated W for Slot 9</td>
<td></td>
</tr>
<tr>
<td>R30.0</td>
<td>31:0</td>
<td>Interpolated W for Slot 8</td>
<td></td>
</tr>
<tr>
<td><strong>R31:</strong> Delivered only if <strong>Position XY Offset Select</strong> is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R31.7</td>
<td>31:24</td>
<td>Position Offset Y for Slot 15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of <strong>Position XY Offset Select</strong>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0.0,1.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>Position Offset X for Slot 15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field contains either the CENTROID or SAMPLE position offset for X, depending on the state of <strong>Position XY Offset Select</strong>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0.0,1.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Position Offset Y for Slot 14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Position Offset X for Slot 14</td>
<td></td>
</tr>
<tr>
<td>R31.6</td>
<td>31:24</td>
<td>Position Offset Y for Slot 13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>Position Offset X for Slot 13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Position Offset Y for Slot 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td>Position Offset X for Slot 12</td>
<td></td>
</tr>
<tr>
<td>R31.5:4</td>
<td>Position Offset X/Y for Slot[11:8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R31.3:2</td>
<td>Position Offset X/Y for Slot[7:4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R31.1:0</td>
<td>Position Offset X/Y for Slot[3:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R32: Delivered only if <strong>Pixel Shader Uses Input Coverage Mask</strong> is set.</td>
<td></td>
</tr>
<tr>
<td>R32.7</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase is only included if <strong>Pixel Shader Uses Input Coverage Mask</strong> (3DSTATE_PS) is set.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field always encodes sample Coverage Mask.</td>
<td></td>
</tr>
<tr>
<td>R32.6</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 6</td>
<td></td>
</tr>
<tr>
<td>R32.5</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 5</td>
<td></td>
</tr>
<tr>
<td>R32.4</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 4</td>
<td></td>
</tr>
<tr>
<td>R32.3</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 3</td>
<td></td>
</tr>
<tr>
<td>R32.2</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 2</td>
<td></td>
</tr>
<tr>
<td>R32.1</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 1</td>
<td></td>
</tr>
<tr>
<td>R32.0</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R33: Delivered only if <strong>Pixel Shader Uses Input Coverage Mask</strong> is set and this is not an 8-pixel dispatch.</td>
<td></td>
</tr>
<tr>
<td>R33.7</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 15</td>
<td></td>
</tr>
<tr>
<td>R33.6</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 14</td>
<td></td>
</tr>
<tr>
<td>R33.5</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 13</td>
<td></td>
</tr>
<tr>
<td>R33.4</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 12</td>
<td></td>
</tr>
<tr>
<td>R33.3</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 11</td>
<td></td>
</tr>
<tr>
<td>R33.2</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 10</td>
<td></td>
</tr>
<tr>
<td>R33.1</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 9</td>
<td></td>
</tr>
<tr>
<td>R33.0</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R34-R57: Delivered only if the corresponding <strong>Barycentric Interpolation Mode</strong> bit is set and this is a 32-pixel dispatch.</td>
<td></td>
</tr>
<tr>
<td>R34.7</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase is only included if the corresponding enable bit in <strong>Barycentric Interpolation Mode</strong> is set.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td>R34.6</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 22</td>
<td></td>
</tr>
<tr>
<td>R34.5</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 21</td>
<td></td>
</tr>
<tr>
<td>R34.4</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 20</td>
<td></td>
</tr>
<tr>
<td>R34.3</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 19</td>
<td></td>
</tr>
<tr>
<td>R34.2</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 18</td>
<td></td>
</tr>
<tr>
<td>R34.1</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 17</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>R34.0</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 16</td>
<td></td>
</tr>
<tr>
<td>R36.7</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 31</td>
<td></td>
</tr>
<tr>
<td>R36.6</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 30</td>
<td></td>
</tr>
<tr>
<td>R36.5</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 29</td>
<td></td>
</tr>
<tr>
<td>R36.4</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 28</td>
<td></td>
</tr>
<tr>
<td>R36.3</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 27</td>
<td></td>
</tr>
<tr>
<td>R36.2</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 26</td>
<td></td>
</tr>
<tr>
<td>R36.1</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 25</td>
<td></td>
</tr>
<tr>
<td>R36.0</td>
<td>31:0</td>
<td>Perspective Pixel Location Barycentric[1] for Slot 24</td>
<td></td>
</tr>
<tr>
<td>R38.41</td>
<td></td>
<td>Perspective Centroid Barycentric</td>
<td></td>
</tr>
<tr>
<td>R42.45</td>
<td></td>
<td>Perspective Sample Barycentric</td>
<td></td>
</tr>
<tr>
<td>R46.49</td>
<td></td>
<td>Linear Pixel Location Barycentric</td>
<td></td>
</tr>
<tr>
<td>R50.53</td>
<td></td>
<td>Linear Centroid Barycentric</td>
<td></td>
</tr>
<tr>
<td>R54.57</td>
<td></td>
<td>Linear Sample Barycentric</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R58-R59:</strong> Delivered only if <strong>Pixel Shader Uses Source Depth</strong> is set and this is a 32-pixel dispatch.</td>
<td></td>
</tr>
<tr>
<td>R58.7</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase is only included if <strong>Pixel Shader Uses Source Depth (WM_STATE)</strong> bit is set.</td>
<td></td>
</tr>
<tr>
<td>R58.6</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 22</td>
<td></td>
</tr>
<tr>
<td>R58.5</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 21</td>
<td></td>
</tr>
<tr>
<td>R58.4</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 20</td>
<td></td>
</tr>
<tr>
<td>R58.3</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 19</td>
<td></td>
</tr>
<tr>
<td>R58.2</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 18</td>
<td></td>
</tr>
<tr>
<td>R58.1</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 17</td>
<td></td>
</tr>
<tr>
<td>R58.0</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 16</td>
<td></td>
</tr>
<tr>
<td>R59.7</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 31</td>
<td></td>
</tr>
<tr>
<td>R59.6</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 30</td>
<td></td>
</tr>
<tr>
<td>R59.5</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 29</td>
<td></td>
</tr>
<tr>
<td>R59.4</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 28</td>
<td></td>
</tr>
<tr>
<td>R59.3</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 27</td>
<td></td>
</tr>
<tr>
<td>R59.2</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 26</td>
<td></td>
</tr>
<tr>
<td>R59.1</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 25</td>
<td></td>
</tr>
<tr>
<td>R59.0</td>
<td>31:0</td>
<td>Interpolated Depth for Slot 24</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>R60-R61:</strong> Delivered only if <strong>Pixel Shader Uses Source W</strong> is set and this is</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>R60.7</td>
<td>31:0</td>
<td>Interpolated W for Slot 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase are only included if <strong>Pixel Shader Uses</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Source W</strong> (WM_STATE) bit is set.</td>
<td></td>
</tr>
<tr>
<td>R60.6</td>
<td>31:0</td>
<td>Interpolated W for Slot 22</td>
<td></td>
</tr>
<tr>
<td>R60.5</td>
<td>31:0</td>
<td>Interpolated W for Slot 21</td>
<td></td>
</tr>
<tr>
<td>R60.4</td>
<td>31:0</td>
<td>Interpolated W for Slot 20</td>
<td></td>
</tr>
<tr>
<td>R60.3</td>
<td>31:0</td>
<td>Interpolated W for Slot 19</td>
<td></td>
</tr>
<tr>
<td>R60.2</td>
<td>31:0</td>
<td>Interpolated W for Slot 18</td>
<td></td>
</tr>
<tr>
<td>R60.1</td>
<td>31:0</td>
<td>Interpolated W for Slot 17</td>
<td></td>
</tr>
<tr>
<td>R60.0</td>
<td>31:0</td>
<td>Interpolated W for Slot 16</td>
<td></td>
</tr>
<tr>
<td>R61.7</td>
<td>31:0</td>
<td>Interpolated W for Slot 31</td>
<td></td>
</tr>
<tr>
<td>R61.6</td>
<td>31:0</td>
<td>Interpolated W for Slot 30</td>
<td></td>
</tr>
<tr>
<td>R61.5</td>
<td>31:0</td>
<td>Interpolated W for Slot 29</td>
<td></td>
</tr>
<tr>
<td>R61.4</td>
<td>31:0</td>
<td>Interpolated W for Slot 28</td>
<td></td>
</tr>
<tr>
<td>R61.3</td>
<td>31:0</td>
<td>Interpolated W for Slot 27</td>
<td></td>
</tr>
<tr>
<td>R61.2</td>
<td>31:0</td>
<td>Interpolated W for Slot 26</td>
<td></td>
</tr>
<tr>
<td>R61.1</td>
<td>31:0</td>
<td>Interpolated W for Slot 25</td>
<td></td>
</tr>
<tr>
<td>R61.0</td>
<td>31:0</td>
<td>Interpolated W for Slot 24</td>
<td></td>
</tr>
</tbody>
</table>

| 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                                              |

**R62**: Delivered only if **Position XY Offset Select** is either **POSOFFSET_CENTROID** or **POSOFFSET_SAMPLE** and this is a **32-pixel dispatch**.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Position Offset Y for Slot 28</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Position Offset X for Slot 28</strong></td>
<td></td>
</tr>
<tr>
<td>R62.5:4</td>
<td>7:0</td>
<td>Position Offset X/Y for Slot[27:24]</td>
<td></td>
</tr>
<tr>
<td>R62.3:2</td>
<td>7:0</td>
<td>Position Offset X/Y for Slot[23:20]</td>
<td></td>
</tr>
<tr>
<td>R62.1:0</td>
<td>7:0</td>
<td>Position Offset X/Y for Slot[19:16]</td>
<td></td>
</tr>
<tr>
<td><strong>R63-R64</strong>: Delivered only if <strong>Pixel Shader Uses Input Coverage Mask</strong> is set and this is a 32-pixel dispatch.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R63.7</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = U32</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This and the next register phase are only included if <strong>Pixel Shader Uses Input Coverage Mask</strong> (<strong>3DSTATE_PS</strong>) is set.</td>
<td></td>
</tr>
<tr>
<td>R63.6</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 22</td>
<td></td>
</tr>
<tr>
<td>R63.5</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 21</td>
<td></td>
</tr>
<tr>
<td>R63.4</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 20</td>
<td></td>
</tr>
<tr>
<td>R63.3</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 19</td>
<td></td>
</tr>
<tr>
<td>R63.2</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 18</td>
<td></td>
</tr>
<tr>
<td>R63.1</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 17</td>
<td></td>
</tr>
<tr>
<td>R63.0</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 16</td>
<td></td>
</tr>
<tr>
<td>R64.7</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 31</td>
<td></td>
</tr>
<tr>
<td>R64.6</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 30</td>
<td></td>
</tr>
<tr>
<td>R64.5</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 29</td>
<td></td>
</tr>
<tr>
<td>R64.4</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 28</td>
<td></td>
</tr>
<tr>
<td>R64.3</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 27</td>
<td></td>
</tr>
<tr>
<td>R64.2</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 26</td>
<td></td>
</tr>
<tr>
<td>R64.1</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 25</td>
<td></td>
</tr>
<tr>
<td>R64.0</td>
<td>31:0</td>
<td>Input Coverage Mask for Slot 24</td>
<td></td>
</tr>
<tr>
<td><strong>R66</strong>: delivered only if <strong>Pixel Shader Requires Source Depth and/or W Attribute Vertex Deltas</strong> is set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R66.7</td>
<td>31:0</td>
<td><strong>rhw_c0</strong> – Co for 1/w plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = IEEE_Float</strong></td>
<td></td>
</tr>
<tr>
<td>R66.6</td>
<td>31:0</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td>R66.5</td>
<td>31:0</td>
<td><strong>rhw_cx</strong> – Cx for 1/w plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = IEEE_Float</strong></td>
<td></td>
</tr>
<tr>
<td>R66.4</td>
<td>31:0</td>
<td><strong>rhw_cy</strong> - Cy for 1/w plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Format = IEEE_Float</strong></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| R66.3 | 31:0 | z_c0 – Co for z plane  
Format = IEEE_Float | |
| R66.2 | 31:0 | Reserved – MBZ | |
| R66.1 | 31:0 | z_cx – Cx for z plane  
Format = IEEE_Float | |
| R66.0 | 31:0 | z_cy – Cy for z plane  
Format = IEEE_Float | |
| R67.7 | 31:0 | bary2_c0 – Co for bary2/w plane  
Format = IEEE_Float | |
| R67.6 | 31:0 | Reserved – MBZ | |
| R67.5 | 31:0 | bary2_cx – Cx for bary2/w plane  
Format = IEEE_Float | |
| R67.4 | 31:0 | bary2_cy – Cy for bary2/w plane  
Format = IEEE_Float | |
| R67.3 | 31:0 | bary1_c0 – Co for bary1/w plane  
Format = IEEE_Float | |
| R67.2 | 31:0 | Reserved – MBZ | |
| R67.1 | 31:0 | bary1_cx – Cx for bary1/w plane  
Format = IEEE_Float | |
| R67.0 | 31:0 | bary1_cy – Cy for bary1/w plane  
Format = IEEE_Float | |
| R68.7 | 31:0 | npc_bary2_c0 – Co for npc_bary2 plane  
Format = IEEE_Float | |
| R68.6 | 31:0 | Reserved – MBZ | |
| R68.5 | 31:0 | npc_bary2_cx – Cx for npc_bary2 plane  
Format = IEEE_Float | |

**R67:** delivered only if **Pixel Shader Requires Perspective Bary Planes** is set.

**R68:** delivered only if **Pixel Shader Requires Non-Perspective Bary Planes** is set.
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>R68.4</td>
<td>31:0</td>
<td><code>npc_bary2_cy</code> - Cy for <code>npc_bary2</code> plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td>R68.3</td>
<td>31:0</td>
<td><code>npc_bary1_c0</code> - Co for <code>npc_bary1</code> plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td>R68.2</td>
<td>31:0</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td>R68.1</td>
<td>31:0</td>
<td><code>npc_bary1_cx</code> - Cx for <code>npc_bary1</code> plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
<tr>
<td>R68.0</td>
<td>31:0</td>
<td><code>npc_bary1_cy</code> - Cy for <code>npc_bary1</code> plane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = IEEE_Float</td>
<td></td>
</tr>
</tbody>
</table>

**R69:** delivered only if **Pixel Shader Requires sample offsets** is set.

| R69.7 | 31:28 | Reserved – MBZ                           |         |
|       | 27:24 | Sub-sample Y offset for sample 15        |         |
|       |      | Format: U0.4                             |         |
|       |      | Subpixel Y offset of Sample 15 relative to the UL pixel origin |         |
|       |      | Range: [0,0.9375]                        |         |
|       | 23:20 | Reserved – MBZ                           |         |
|       | 19:16 | Sub-sample Y offset for sample 14        |         |
|       | 15:12 | Reserved – MBZ                           |         |
|       | 11:8  | Sub-sample Y offset for sample 13        |         |
|       | 7:4   | Reserved – MBZ                           |         |
|       | 3:0   | Sub-sample Y offset for sample 12        |         |

| R69.6 | 31:28 | Reserved – MBZ                           |         |
|       | 27:24 | Sub-sample Y offset for sample 11        |         |
|       | 23:20 | Reserved – MBZ                           |         |
|       | 19:16 | Sub-sample Y offset for sample 10        |         |
|       | 15:12 | Reserved – MBZ                           |         |
|       | 11:8  | Sub-sample Y offset for sample 9         |         |
|       | 7:4   | Reserved – MBZ                           |         |
|       | 3:0   | Sub-sample Y offset for sample 8         |         |

<p>| R69.5 | 31:28 | Reserved – MBZ                           |         |
|       | 27:24 | Sub-sample Y offset for sample 7         |         |
|       | 23:20 | Reserved – MBZ                           |         |
|       | 19:16 | Sub-sample Y offset for sample 6         |         |
|       | 15:12 | Reserved – MBZ                           |         |</p>
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample Y offset for sample 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample Y offset for sample 4</td>
<td></td>
</tr>
<tr>
<td>R69.4</td>
<td>31:28</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Sub-sample Y offset for sample 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Sub-sample Y offset for sample 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample Y offset for sample 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample Y offset for sample 0</td>
<td></td>
</tr>
<tr>
<td>R69.3</td>
<td>31:28</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Sub-sample X offset for sample 15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U0.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subpixel X offset of Sample 15 relative to the UL pixel origin</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range: [0,0.9375]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Sub-sample X offset for sample 14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample X offset for sample 13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample X offset for sample 12</td>
<td></td>
</tr>
<tr>
<td>R69.2</td>
<td>31:28</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Sub-sample X offset for sample 11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Sub-sample X offset for sample 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample X offset for sample 9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample X offset for sample 8</td>
<td></td>
</tr>
<tr>
<td>R69.1</td>
<td>31:28</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Sub-sample X offset for sample 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Sub-sample X offset for sample 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample X offset for sample 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample X offset for sample 4</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>R69.0</td>
<td>31:28</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>Sub-sample X offset for sample 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19:16</td>
<td>Sub-sample X offset for sample 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:12</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11:8</td>
<td>Sub-sample X offset for sample 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:4</td>
<td>Reserved – MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td>Sub-sample X offset for sample 0</td>
<td></td>
</tr>
</tbody>
</table>

**Optional Padding before the Start of Constant/Setup Data**

The locations between the end of the Optional Payload Header and the location programmed via Dispatch GRF Start Register for Constant/Setup Data are considered "padding" and Reserved (see below).

**Optional, multiple of 8 DWs**

| 31:0      | Reserved                                          |

The **Dispatch GRF Start Register for Constant/Setup Data** state variable in 3DSTATE_WM is used to define the starting location of the constant and setup data within the PS thread payload. This control is provided to allow this data to be located at a fixed location within thread payloads, regardless of the amount of data in the Optional Payload Header. This permits the kernel to use direct GRF addressing to access the constant/setup data, regardless of the optional parameters being passed (as these are determined on-the-fly by the WM unit).

**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_PS command (taking the buffer enables into account).

The Constant Data arrives in a non-interleaved format.

**Optional, multiple of 8 DWs**

| 31:0 | **Constant Data** |

**Setup Data** (Attribute Vertex Deltas)

Output data from the SF stage is delivered in the PS thread payload. The amount of data is determined by the **Number of Output Attributes** field. Each register contains two channels of one attribute. Thus, the total number of registers sent is equal to 2 * Number of Output Attributes.

<table>
<thead>
<tr>
<th>Rp.7</th>
<th>31:0</th>
<th>a0[0].y – a0 vertex data for Attribute0.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rp.6</td>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>Rp.5</td>
<td>31:0</td>
<td>a2[0].y – a2-a0 vertex delta for Attribute0.y</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Rp.4</td>
<td>31:0</td>
<td>\texttt{a1[0].y} – \texttt{a1-a0 vertex delta for Attribute0.y}</td>
</tr>
<tr>
<td>Rp.3</td>
<td>31:0</td>
<td>\texttt{a0[0].x} – \texttt{a0 vertex data for Attribute0.x}</td>
</tr>
<tr>
<td>Rp.2</td>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>Rp.1</td>
<td>31:0</td>
<td>\texttt{a2[0].x} – \texttt{a2-a0 vertex delta for Attribute0.x}</td>
</tr>
<tr>
<td>Rp.0</td>
<td>31:0</td>
<td>\texttt{a1[0].x} – \texttt{a1-a0 vertex delta for Attribute0.x}</td>
</tr>
<tr>
<td>R(p+1).7</td>
<td>31:0</td>
<td>\texttt{a0[0].w} – \texttt{a0 vertex data for Attribute0.w}</td>
</tr>
<tr>
<td>R(p+1).6</td>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R(p+1).5</td>
<td>31:0</td>
<td>\texttt{a2[0].w} – \texttt{a2-a0 vertex delta for Attribute0.w}</td>
</tr>
<tr>
<td>R(p+1).4</td>
<td>31:0</td>
<td>\texttt{a1[0].w} – \texttt{a1-a0 vertex delta for Attribute0.w}</td>
</tr>
<tr>
<td>R(p+1).3</td>
<td>31:0</td>
<td>\texttt{a0[0].z} – \texttt{a0 vertex data for Attribute0.z}</td>
</tr>
<tr>
<td>R(p+1).2</td>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>R(p+1).1</td>
<td>31:0</td>
<td>\texttt{a2[0].z} – \texttt{a2-a0 vertex delta for Attribute0.z}</td>
</tr>
<tr>
<td>R(p+1).0</td>
<td>31:0</td>
<td>\texttt{a1[0].z} – \texttt{a1-a0 vertex delta for Attribute0.z}</td>
</tr>
<tr>
<td>R(p+2*n)</td>
<td></td>
<td>\textit{xy Vertex Deltas for Attributes n} \footnote{See definition of Rp for format.}</td>
</tr>
<tr>
<td>R(p+2*n+1)</td>
<td></td>
<td>\textit{zw Vertex Deltas for Attribute n} \footnote{See definition of R(p+1) for format.}</td>
</tr>
</tbody>
</table>
Pixel Backend

This section contains the following subsections:

- Various Color Calculators
- 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.

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha Coverage</td>
<td>It generates coverage masks using AlphaToCoverage AND/OR AlphaToOne functions based on src0.alpha.</td>
</tr>
<tr>
<td>Alpha Test</td>
<td>Compare pixel alpha with reference alpha and conditionally discard pixel.</td>
</tr>
<tr>
<td>Stencil Test</td>
<td>Compare pixel stencil value with reference and forward result to Buffer Update stage.</td>
</tr>
<tr>
<td>Depth Test</td>
<td>Compare pix.Z with corresponding Z value in the Depth Buffer and forward result to Buffer Update stage.</td>
</tr>
<tr>
<td>Color Blending</td>
<td>Combine pixel color with corresponding color in color buffer according to programmable function.</td>
</tr>
<tr>
<td>Gamma Correction</td>
<td>Adjust pixel's color according to gamma function for SRGB destination surfaces.</td>
</tr>
<tr>
<td>Color Quantization</td>
<td>Convert &quot;full precision&quot; pixel color values to fixed precision of the color buffer format.</td>
</tr>
<tr>
<td>Logic Ops</td>
<td>Combine pixel color logically with existing color buffer color (mutually exclusive with Color Blending).</td>
</tr>
<tr>
<td>Buffer Update</td>
<td>Write final pixel values to color and depth buffers or discard pixel without update.</td>
</tr>
</tbody>
</table>

The following logic describes the high-level operation of the Pixel Processing pipeline:

```cpp
PixelProcessing() {
    AlphaCoverage()
    AlphaTest()
    DepthBufferCoordinateOffsetDisable
    StencilTest()
    DepthTest()
    ColorBufferBlending()
    GammaCorrection()
    ColorQuantization()
    LogicalOps()
    BufferUpdate()
}
```
Alpha Coverage

Alpha coverage logic is supported for DevSNB+ 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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>If Pixel Shader outputs oMask, AlphaToCoverage is disabled in hardware, regardless of the state setting for this feature.</td>
<td></td>
</tr>
</tbody>
</table>

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 Restrictions:**

- 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.
**Stencil Test**

The Stencil Test function can be used to discard pixels based on a comparison between the [Backface] Stencil Test Reference state variable and the pixel's stencil value. This is a general purpose function used for such effects as shadow volumes, per-pixel clipping, etc. The result of this comparison is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the Stencil Test Enable state variable. If ENABLED, the current stencil buffer value for this pixel is read.

**Programming Note:**

- If the Depth Buffer is either undefined or does not have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT and separate stencil buffer is disabled, Stencil Test Enable must be DISABLED.

A 2nd set of the stencil test state variables is provided so that pixels from back-facing objects, assuming they are not culled, can have a stencil test performed on them separate from the test for normal front-facing objects. The separate stencil test for back-facing objects can be enabled via the Double Sided Stencil Enable state variable. Otherwise, non-culled back-facing objects will use the same test function, mask and reference value as front-facing objects. The 2nd stencil state for back-facing objects is most commonly used to improve the performance of rendering shadow volumes which require a different stencil buffer operation depending on whether pixels rendered are from a front-facing or back-facing object. The backface stencil state removes the requirement to render the shadow volumes in 2 passes or sort the objects into front-facing and back-facing lists.

The remainder of this subsection describes the function in terms 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.

<table>
<thead>
<tr>
<th>Blending</th>
<th>RT Format</th>
<th>Pre-Blend Color Clamp</th>
<th>Post-Blend Color Clamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>UNORM, UNORM_SRGB, YCRCB</td>
<td>Must be enabled with range = RT range or [0,1] (same function)</td>
<td>N/A, state ignored</td>
</tr>
<tr>
<td></td>
<td>SNORM</td>
<td>Must be enabled with range = RT range or [-1,1] (same function)</td>
<td>N/A, state ignored</td>
</tr>
<tr>
<td></td>
<td>FLOAT (except for R11G11B10_FLOAT)</td>
<td>Must be enabled (with any desired range)</td>
<td>N/A, state ignored</td>
</tr>
<tr>
<td></td>
<td>R11G11B10_FLOAT</td>
<td>Must be enabled with either [0,1] or RT range</td>
<td>N/A, state ignored</td>
</tr>
<tr>
<td></td>
<td>UINT, SINT</td>
<td>State ignored, implied clamp to RT range</td>
<td>N/A, state ignored</td>
</tr>
<tr>
<td>On</td>
<td>UNORM, UNORM_SRGB</td>
<td>Must be enabled with range = RT range or [0,1] (same function)</td>
<td>Must be enabled with range = RT range or [0,1] (same function)</td>
</tr>
</tbody>
</table>
**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 Notes:**

- 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 Notes:**

<table>
<thead>
<tr>
<th>Blending</th>
<th>RT Format</th>
<th>Pre-Blend Color Clamp</th>
<th>Post-Blend Color Clamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>permitted</td>
<td>function</td>
<td>function</td>
<td></td>
</tr>
<tr>
<td>SNORM</td>
<td>Must be enabled with range = RT range or [-1,1] (same function)</td>
<td>Must be enabled with range = RT range or [-1,1] (same function)</td>
<td></td>
</tr>
<tr>
<td>FLOAT (except for R11G11B10_FLOAT)</td>
<td>Can be disabled or enabled (with any desired range)</td>
<td>Must be enabled (with any desired range)</td>
<td></td>
</tr>
<tr>
<td>R11G11B10_FLOAT</td>
<td>Can be disabled or enabled (with any desired range)</td>
<td>Must be enabled with either [0,1] or RT range</td>
<td></td>
</tr>
</tbody>
</table>
• Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
• Dual source blending: The DataPort only supports dual source blending with a SIMD8-style message.
• Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in Sampling Engine. Blending must be disabled on a RenderTarget if blending is not supported.

The incoming source pixel values are modulated by a selected source blend factor, and the possibly gamma-decorrected destination values are modulated by a destination blend factor. These terms are then combined with a blend function. In general:

\[
\text{src\_term} = \text{src\_blend\_factor} \times \text{src\_color} \\
\text{dst\_term} = \text{dst\_blend\_factor} \times \text{dst\_color} \\
\text{color output} = \text{blend\_function}(\text{src\_term}, \text{dst\_term})
\]

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

<table>
<thead>
<tr>
<th>Blend Factor Selection</th>
<th>Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLENDFACTOR_ZERO</td>
<td>0.0, 0.0, 0.0, 0.0</td>
</tr>
<tr>
<td>BLENDFACTOR_ONE</td>
<td>1.0, 1.0, 1.0, 1.0</td>
</tr>
<tr>
<td>BLENDFACTOR_SRC_COLOR</td>
<td>oN.r, oN.g, oN.b, oN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_SRC_COLOR</td>
<td>1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_SRC_ALPHA</td>
<td>oN.a, oN.a, oN.a, oN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_SRC_ALPHA</td>
<td>1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_SRC1_COLOR</td>
<td>o1.r, o1.g, o1.b, o1.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_SRC1_COLOR</td>
<td>1.0-o1.r, 1.0-o1.g, 1.0-o1.b, 1.0-o1.a</td>
</tr>
<tr>
<td>BLENDFACTOR_SRC1_ALPHA</td>
<td>o1.a, o1.a, o1.a, o1.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_SRC1_ALPHA</td>
<td>1.0-o1.a, 1.0-o1.a, 1.0-o1.a, 1.0-o1.a</td>
</tr>
<tr>
<td>BLENDFACTOR_DST_COLOR</td>
<td>rtN.r, rtN.g, rtN.b, rtN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_DST_COLOR</td>
<td>1.0-rtN.r, 1.0-rtN.g, 1.0-rtN.b, 1.0-rtN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_DST_ALPHA</td>
<td>rtN.a, rtN.a, rtN.a, rtN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_DST_ALPHA</td>
<td>1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a</td>
</tr>
<tr>
<td>BLENDFACTOR_CONST_COLOR</td>
<td>CC.r, CC.g, CC.b, CC.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_CONST_COLOR</td>
<td>1.0-CC.r, 1.0-CC.g, 1.0-CC.b, 1.0-CC.a</td>
</tr>
<tr>
<td>BLENDFACTOR_CONST_ALPHA</td>
<td>CC.a, CC.a, CC.a, CC.a</td>
</tr>
<tr>
<td>BLENDFACTOR_INV_CONST_ALPHA</td>
<td>1.0-CC.a, 1.0-CC.a, 1.0-CC.a, 1.0-CC.a</td>
</tr>
<tr>
<td>BLENDFACTOR_SRC_ALPHA_SATURATE</td>
<td>f,f,1.0 where f = min(1.0 – rtN.a, oN.a)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Blend Function</th>
<th>Operation (for each color component)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLENDFUNCTION_ADD</td>
<td>SrcColor<em>SrcFactor + DstColor</em>DstFactor</td>
</tr>
<tr>
<td>BLENDFUNCTION_SUBTRACT</td>
<td>SrcColor<em>SrcFactor - DstColor</em>DstFactor</td>
</tr>
<tr>
<td>BLENDFUNCTION_REVERSE_SUBTRACT</td>
<td>DstColor<em>DstFactor - SrcColor</em>SrcFactor</td>
</tr>
<tr>
<td>BLENDFUNCTION_MIN</td>
<td>min (SrcColor<em>SrcFactor, DstColor</em>DstFactor)</td>
</tr>
<tr>
<td><strong>Programming Note:</strong></td>
<td>This is a superset of the OpenGL min function.</td>
</tr>
<tr>
<td>BLENDFUNCTION_MAX</td>
<td>max (SrcColor<em>SrcFactor, DstColor</em>DstFactor)</td>
</tr>
</tbody>
</table>
**Blend Function**

<table>
<thead>
<tr>
<th>Blend Function</th>
<th>Operation (for each color component)</th>
<th>Programming Note: This is a superset of the OpenGL max function.</th>
</tr>
</thead>
</table>

**Post-Blend Color Clamping**

(See Pre-Blend Color Clamping above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

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

**Dithering**

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer’s eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel’s X and Y screen coordinate. The pixel’s X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel’s screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel’s screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.
Dithering Process (5-Bit Example)

Logic Ops

The Logic Ops function is used to combine the incoming "source" pixel color/alpha values with the corresponding "destination" color/alpha contained in the ColorBuffer, using a logic function.

The Logic Op function is enabled by the LogicOp Enable state variable. If DISABLED, this function is ignored and the incoming pixel values are passed through unchanged.

Programming Notes

<table>
<thead>
<tr>
<th>Project</th>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.</td>
<td></td>
</tr>
<tr>
<td>Logic Ops are supported on all blendable render targets and render targets with *INT formats.</td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>LogicOp Function</th>
<th>Definition (S=Source, D=Destination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICOP_CLEAR</td>
<td>all 0’s</td>
</tr>
<tr>
<td>LOGICOP_NOR</td>
<td>NOT (S OR D)</td>
</tr>
<tr>
<td>LOGICOP_AND_INVERTED</td>
<td>(NOT S) AND D</td>
</tr>
<tr>
<td>LOGICOP_COPY_INVERTED</td>
<td>NOT S</td>
</tr>
<tr>
<td>LOGICOP_AND_REVERSE</td>
<td>S AND NOT D</td>
</tr>
<tr>
<td>LOGICOP_INVERT</td>
<td>NOT D</td>
</tr>
<tr>
<td>LOGICOP_XOR</td>
<td>S XOR D</td>
</tr>
<tr>
<td>LOGICOP_NAND</td>
<td>NOT (S AND D)</td>
</tr>
<tr>
<td>LOGICOP_AND</td>
<td>S AND D</td>
</tr>
<tr>
<td>LOGICOP_EQUIV</td>
<td>NOT (S XOR D)</td>
</tr>
<tr>
<td>LOGICOP_NOOP</td>
<td>D</td>
</tr>
<tr>
<td>LOGICOP_OR_INVERTED</td>
<td>(NOT S) OR D</td>
</tr>
<tr>
<td>LOGICOP_COPY</td>
<td>S</td>
</tr>
<tr>
<td>LOGICOP_OR_REVERSE</td>
<td>S OR NOT D</td>
</tr>
<tr>
<td>LOGICOP_OR</td>
<td>S OR D</td>
</tr>
<tr>
<td>LOGICOP_SET</td>
<td>all 1’s</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Stencil Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCILOP_KEEP</td>
<td>Do not modify the stencil buffer</td>
</tr>
<tr>
<td>STENCILOP_ZERO</td>
<td>Store a 0</td>
</tr>
<tr>
<td>STENCILOP_REPLACE</td>
<td>Store the StencilTestReference reference value</td>
</tr>
<tr>
<td>STENCILOP_INCRSAT</td>
<td>Saturating increment (clamp to max value)</td>
</tr>
<tr>
<td>STENCILOP_DECRSAT</td>
<td>Saturating decrement (clamp to 0)</td>
</tr>
<tr>
<td>STENCILOP_INCR</td>
<td>Increment (possible wrap around to 0)</td>
</tr>
<tr>
<td>STENCILOP_DECR</td>
<td>Decrement (possible wrap to max value)</td>
</tr>
<tr>
<td>STENCILOP_INVERT</td>
<td>Logically invert the stencil value</td>
</tr>
</tbody>
</table>

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 Notes:
- The Stencil Buffer can be written even if depth buffer writes are disabled via Depth Buffer Write Enable.

## Depth Buffer Updates

Any and all writes to the Depth Buffer are enabled by the Depth Buffer Write Enable state variable. If there is no Depth Buffer, writes must be explicitly disabled with this state variable, or operation is UNDEFINED.

If depth testing is disabled or the depth test passed, the incoming pixel's depth value is written to the Depth Buffer. If depth testing is enabled and the depth test failed, the pixel is discarded – with no modification to the Depth or Color Buffers (though the Stencil Buffer may have been modified).

## Color Gamma Correction

Computed RGB (not A) channels can be gamma-corrected prior to update of the Color Buffer.

This function is automatically invoked whenever the destination surface (render target) has an SRGB format (see surface formats in Sampling Engine). For these surfaces, the computed RGB values are converted from gamma=1.0 space to gamma=2.4 space by applying a \(^{(2.4)}\) exponential function.
Color Buffer Updates

Finally, if the pixel has not been discarded by this point, the incoming pixel color is written into the Color Buffer. The Surface Format of the color buffer indicates which channel(s) are written (e.g., R8G8_UNORM are written with the Red and Green channels only). The Color Buffer Component Write Disables from the Color buffer’s SURFACE_STATE provide an independent write disable for each channel of the Color Buffer.

Pixel Pipeline State Summary

COLOR_CALC_STATE

3DSTATE_BLEND_STATE_POINTERS

3DSTATE_DEPTH_STENCIL_STATE_POINTERS

COLOR_CALC_STATE

COLOR_CALC_STATE

DEPTH_STENCIL_STATE

DEPTH_STENCIL_STATE

BLEND_STATE

BLEND_STATE

Programming Note: CC Unit also receives 3DSTATE_WM_HZ_OP and 3DSTATE_PS_EXTRA.

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)**

<table>
<thead>
<tr>
<th>Cache Mode MMIO Bit (Please refer to Vol 1c)</th>
<th>MSC Enable (Surface State)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (feature disable)</td>
<td>X</td>
<td>Normal mode of operation i.e. no MSAA compression and no color clear</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Normal mode of operation i.e. no MSAA compression and no color clear</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Depending on the Number of multi-samples, either MSAA compression OR color clear is enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>MSAA</th>
<th>Width of Clear Rect</th>
<th>Height of Clear Rect</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>4X</td>
<td>Ceil(1/8*width)</td>
<td>Ceil(1/2*height)</td>
</tr>
<tr>
<td>HSW</td>
<td>8X</td>
<td>Ceil(1/2*width)</td>
<td>Ceil(1/2*height)</td>
</tr>
</tbody>
</table>

- **MSAA Compression:** Multi-sample render target is bound to the pipeline and MSAA compression feature is enabled. In this case, MCS buffer stores the information required for MSAA compression algorithm. The size and layout of the MCS buffer is based on per-pixel RT. For 4X and 8X MSAA, MCS buffer element is 8bpp and 32bpp respectively. Height, width, and layout of MCS buffer in this case must match with Render Target height, width, and layout. MCS buffer is tiledY. When MCS buffer is enabled and bound to MSRT, it is required that it is cleared prior to any rendering. A clear value can be specified optionally in the surface state of the corresponding RT. Clear pass for this case requires that scaled down primitive is sent down with upper left coordinate to coincide with actual rectangle being cleared. For MSAA, clear rectangle's height and width need to as show in the following table in terms of (width, height) of the RT.

- **Fast Color Clear:** When non multi-sample render target is bond to the pipeline and MSC buffer is enabled, MCS buffer is used as an intermediate (coarse granular) buffer per RT. Hence, MCS buffer
is used to improve render target clear. When MCS is buffer is used for color clear of non-multisampler render target, the following restrictions apply:

**Color Clear of Non-MultiSampler Render Target Restrictions**

<table>
<thead>
<tr>
<th>Project</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support is limited to tiled render targets.</td>
<td></td>
</tr>
<tr>
<td>HSW</td>
<td>Support is for non-mip-mapped and non-array surface types only.</td>
</tr>
<tr>
<td>Clear is supported only on the full RT; i.e., no partial clear or overlapping clears.</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the RT alignment:

<table>
<thead>
<tr>
<th>TiledY RT CL</th>
<th>Pixels</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TiledX RT CL</th>
<th>Pixels</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pixels</td>
</tr>
<tr>
<td><strong>TiledY RT</strong></td>
<td></td>
</tr>
<tr>
<td>bpp</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td><strong>TiledX RT</strong></td>
<td></td>
</tr>
<tr>
<td>bpp</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Horizontal Scale Down Factor</td>
</tr>
<tr>
<td><strong>MCS CL for TiledY RCC</strong></td>
<td></td>
</tr>
<tr>
<td>bpp</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td><strong>MCS CL for TiledX RCC</strong></td>
<td></td>
</tr>
<tr>
<td>bpp</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are the following added requirements when performing a render target resolve.

A rectangle primitive must be scaled down by the following factors with respect to render target being resolved.

<table>
<thead>
<tr>
<th>Resolve rectangle scaling for TiledY RCC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>width scale down factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>height scale down factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bpp</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resolve rectangle scaling for TiledX RCC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bpp</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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).

**Note:** Depth Test Enable, Depth Buffer Write Enable, Stencil Test Enable, Stencil Buffer Write Enable, and Alpha Test Enable must all be disabled.

**Note:** This render target resolve procedure is not supported on multisampled render targets. Unresolved multisampled render targets are directly supported by the sampling engine, which resolves clear values in addition to decompressing the surface. This applies to both ld2dms and sample2dms messages.
L3/URB

This section discusses GFX L3 cache. The included topics are:

- Overview
- Atomics
- L3 Coherency
- L3 Allocation & Programming
- L3 Interfaces
- State Arbiter
- L3 Invalidation and Flush Flows
- Shared Local Memory (SLM)
- Dynamic Parity Feature for GFX L3 Cache
- L3 Register Space

L3$/URB

GFX L3 cache is introduced for Gen7 GFX core as 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 for EUs/ROWs.

In order to provide the bandwidth needed L3 has been separated into 4x128KB structures which can be accessed concurrently. A 2x clocking is introduced to further enhance the bandwidth and cover the limitations of SRAM (6T) design.

- Formed as 4 (2 for GT1) individual banks each with 128KB in size
- Each logical bank consists of
  - Data Array
  - Tag Array
  - LRU Array (implements a Pseudo Least Recently Used algorithm)
  - State Array
  - SuperQ Data Buffer
  - Atomic Processing Unit
- 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 operates in GFX coherent domain
• A portion of L3 can be allocated as highly banked memory

**L3$ Cache Configuration**

• 4x128KB cache, 64 logical ways (per slice)
• 64B Cacheline with a portion capable of highly banked memory (with 16x4B capability)
• Interface 64B to SQDB for the fill/write path, 64B Read/Evict path to SQDB
• Data Array built via 6T cells
  - Data protection via parity
• TAG/LRU/STATE (using gen-ram via RLS flows)
  - 32-bit GFX addressing support in TAG
  - 2 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)

**Memory Object Control State on Cacheability**

This 4-bit field is used in various state commands and indirect state objects to define MLC/LLC cacheability and graphics data type for memory objects. For details of the field see the GPU Overview section.

**Atomics**

An atomic operation may involve both reading from and then writing to a memory location. Atomic operations apply only to either u# (Unordered Access Views) or g# (Thread Group Shared Memory). It is guaranteed that when a thread issues an atomic operation on a memory address, no write to the same address from outside the current atomic operation by any thread can occur between the atomic read and write.

If multiple atomic operations from different threads target the same address, the operations are serialized in an undefined order. This serialization happens outside of the L3 control logic.

Atomic operations do not imply a memory or thread fence. If the program author/compiler does not make appropriate use of fences, it is not guaranteed that all threads see the result of any given memory operation at the same time, or in any particular order with respect to updates to other memory addresses.
Atomicity is implemented at 32-bit granularity. If a load or store operation spans more than 32-bits, the individual 32-bit operations are atomic, but not the whole.

**Limitation:** Atomic operations on Thread Group Shared Memory are atomic with respect to other atomic operations, as well as operations that only perform reads ("load"s). However atomic operations on Thread Group Shared Memory are NOT atomic with respect to operations that perform only writes ("store"s) to memory. Mixing of atomics and stores on the same Thread Group Shared Memory address without thread synchronization and memory fencing between them produces undefined results at the address involved. This limitation arises because some implementations of loads and stores do not honor the locking semantics for implementing atomics. It turns out this has no impact on loads, since they are guaranteed to retrieve a value either before or after an atomic (they will not retrieve partially updated values, given they are all defined at 32-bit quanta). However store operations could find their way into the middle of an atomic operation and thus have their effect possibly lost.

In L3 or SLM, the atomic operation leads to a read-modify-write operation on the destination location with the option of returning value back to requester. The table below is defined as a list of atomic operations needed:

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>Description</th>
<th>New Destination Value</th>
<th>Return Value (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic_AND</td>
<td>Single component 32-bit bitwise AND of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>&quot;old_dst&quot; AND &quot;src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_OR</td>
<td>Single component 32-bit bitwise OR of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>&quot;old_dst&quot; OR &quot;src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_XOR</td>
<td>Single component 32-bit bitwise XOR of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>&quot;old_dst&quot; XOR &quot;src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_MOVE</td>
<td>Replacement of the dst with src0.</td>
<td>&quot;src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_INC</td>
<td>Single component 32-bit integer increment of dst back into dst</td>
<td>&quot;old_dst + 1&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_DEC</td>
<td>Single component 32-bit integer decrement of dst back into dst</td>
<td>&quot;old_dst - 1&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_ADD</td>
<td>Single component 32-bit integer add of operand src0 into dst at 32-bit per component address performed atomically. Insensitive to sign</td>
<td>&quot;old_dst + src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_SUB</td>
<td>Single component 32-bit integer subtraction of operand src0 into dst at 32-bit per component address performed atomically. Insensitive to sign</td>
<td>&quot;old_dst - src0&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_RSUB</td>
<td>Single component 32-bit integer subtraction of operand dst from src0 into dst at 32-bit per component address performed atomically. Insensitive to sign</td>
<td>&quot;src0 - old_dst&quot;</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic Operation</td>
<td>Description</td>
<td>New Destination Value</td>
<td>Return Value (optional)</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Atomic_IMAX</td>
<td>Single component 32-bit signed MAX of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>IMAX (old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_IMIN</td>
<td>Single component 32-bit signed MIN of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>IMIN (old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_UMAX</td>
<td>Single component 32-bit unsigned MAX of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>UMAX (old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_UMIN</td>
<td>Single component 32-bit unsigned MIN of operand src0 into dst at 32-bit per component address dstAddress, performed atomically.</td>
<td>UMIN (old_dst, src0)</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_CMP/WR</td>
<td>Single component 32-bit value compare of operand src0 with dst at 32-bit per component address dstAddress. If the compared values are identical, the single-component 32-bit value in src1 is written to destination memory, else the destination is not changed. The entire compare+write operation is performed atomically.</td>
<td>(src0 == old_dst)? src1: old_dst</td>
<td>old_dst</td>
</tr>
<tr>
<td>Atomic_PREDEC</td>
<td>Single component 32-bit integer decrement of dst back into dst</td>
<td>&quot;old_dst - 1&quot;</td>
<td>new_dst</td>
</tr>
<tr>
<td>Atomic_CMP/WR8B</td>
<td>Single component 64-bit value compare of operand src0 with dst at 64-bit per component address dstAddress. If the compared values are identical, the single-component 64-bit value in src1 is written to destination memory, else the destination is not changed. The entire compare+write operation is performed atomically.</td>
<td>(src0 == old_dst)? src1: old_dst</td>
<td>old_dst</td>
</tr>
</tbody>
</table>

The DC request for atomic will have the proper DW only byte enables set for the 32 bit of interest. The address down to bit[2] (dword address) will also be provided to point to correct DW out of 16 lanes in 64Bytes.

The processing of atomics will follow 2 separate pipelines of operation (either SLM or L3) depending on the destination of the access.

Atomics when disabled in L3, are performed at the GFX interface. Non-L3 atomics are not going to be as performant and requires x9030[3:2]="01" to operate.
Atomics in L3

Atomics in L3 are handled separately in each bank, to achieve this function 2 Atomics blocks are instantiated along with each bank. Each operand being moved to SQ also moves its data (up to 2 DWs) into an assigned atomics block to be used later on (when he destination data is available).

A separate credit is given to L3 arbiter for atomics, once an atomic request is moved from L3 arbiter to SQ – both the SQ credits and Atomics credits need to be deducted to regulate the number of atomic requests in SQ. For GT2, this process allows upto 8 atomics to be performed in a given clock.

The request interface allows only 1 DW of atomics per request, data from DC will be given on DW0 (also in DW1 if src1 is given) for all atomic operations regardless of the address of byte enables. Cacheline address will be provided on the interface with proper Byte Enables singling the DW location of the destination.

If final data is returned to client (optional), the DW of interest will be given in the same position pointed out by byte enables (in fact the same DW will be replicated over 16 positions).

Atomics in SLM

SLM pipeline has a mechanism to handle atomics similar to L3/URB pipeline. There is only 1 ALU per SLM subbank. The protocol between DC and L3 allows one atomics to be performed at a given time, the SLM controller will stall the interface if needed. Per atomics request from the DC, only ONE DW can be active on one SLM bank. SLM pipeline can execute b2b atomics request (1 every 1x clock) as long as b2b operations do not conflict on the same bank. If conflict is detected a single clock of bubble is inserted into pipeline in order to update the corresponding bank with SLM output before next operation can be performed (see SLM pipeline details).

Data from DC will be always given on DW0 and DW (if needed) and VALIDs will point to the bank of interest out of 16 banks of SLM. Correct set of byte enables should be provided which is active for the valid bank.

DW of interest is returned to DC on the byte enable corresponding lane of the cacheline.

Atomics in URB

Simple atomics are possible to be processed for URB locations as well. The process should fall out from the L3 path of the atomics and is restricted similar to L3 atomics.

L3 Allocation & Programming

L3 Cache allocation is done on a per way basis which should be consistent across all 4 banks (2 banks for GT1). The way allocation between URB and any of the L3 clients can only be changed post pipeline flush where L3 contains no data. This is required for stream based flushes to be dependent on the way allocation of these corresponding streams. S/W should not be removing ways under a particular stream and expect a later pipelined stream flush to target all the corresponding locations. The stream based
flush will be performed on the existing way allocation of that stream, there is no history of previous way allocation tracked in the hardware.

L3 Cache has been divided into following client pools:

- **Shared Local Memory**: When enabled its size is always fixed to 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 is 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 w/o hot spoting 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 s/w programming models. The programming options in the following table represents 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.

Only the following configurations are allowed for programming.

**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.
Addtional Supported Configuration:

The configuration for \( \{\text{SLM} = 0, \text{URB} = 224, \text{DC} = 32, \text{RO} = 256, \text{IS} = 0, \text{C} = 0, \text{T} = 0, \text{SUM} = 512\} \) was validated as a later supported configuration and can be utilized if desired.

**SLM Mode Allocation**

With the existence of Shared Local Memory, a 64KB chunk from each of the 2 L3 banks will be reserved for SLM usage. The remaining cache space is divided between the remaining clients. SLM allocation is done via reducing the number of ways on the two banks from 64 to 32.

<table>
<thead>
<tr>
<th>Normal Bank</th>
<th>SLM</th>
<th>URB</th>
<th>Rest</th>
<th>DC</th>
<th>RO</th>
<th>I/S</th>
<th>C</th>
<th>T</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>0</td>
<td>512</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>128</td>
<td>128</td>
<td>0</td>
<td>0</td>
<td>512</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>224</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>32</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>224</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>224</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>224</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>32</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>32</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

Given the reduction on the 2 banks for L3, we have a unique problem of how to manage HASH between 4 un-equal size banks. The way to address that issue is to identify "Low Bandwidth" clients and allocate them into un-even ways of the large banks and handle them via a 2-way HASH. The remaining clients are allocated between 4 banks with the equal number of ways on each bank. The shaded clients in corresponding table are supposed to correspond to low bandwidth clients and their total should be 128KB.

Note that, having mixture of 2 hashes need to ensure "there are no coherency requirements between the high b/w and low b/w clients". This solution prevents any producer/consumer models cross groups.
within L3. When programming the low b/w vs high b/w client profile, the need for coherency with the L3 fabric has to be considered.

Note that URB needs to be set as low b/w client in SLM mode, else the hash will fail. This is a required s/w model.

**Additional Supported Configuration:**

The configuration for \{SLM = 128, URB = 128, DC = 0, RO = 256, IS = 0, C = 0, T = 0, SUM 512\} was validated as a later supported configuration and can be utilized if desired. For this configuration, global atomics must be programmed to be in GTI.

**L3 Invalidation and Flush Flows**

**Read Only Stream Invalidations**

All read-only stream invalidations are done at the TOP of the pipe and communicated to L3 fabric directly from the main command streamer. Once send to L3, command streamer can kick off the next workload to reuse the same memory. There are four type streams that can be covered individually or overlapped:

- Texture Invalidation
- Instruction Invalidation
- Constant Invalidation
- State Invalidation

For all types of invalidations the flows are the same from L3 arbiter perspective: first invalidation will directly come from the command streamer and the "state invalidation" is sent via state arbiter unit.

L3 arbiter will propagate the invalidation only after the corresponding streams' requests are retired to superQs. Once a particular invalidation is received, L3 arbiter will put a marker where all the existing requests from that stream. This will allow the already existing requests to be sent to superQ while accepting new requests, however new requests should never be sent to SuperQ until the existing invalidation is complete. Once all existing (marked requests) moved to SQ, L3 arbiter will propagate the invalidation request to SuperQ and keep the corresponding ingress FIFOs blocked.

**Note:** There may be two sources of the corresponding stream (i.e. Textures from either half-slice arbiters...). L3 arbiter needs to ensure both streams are serviced before propagating the "invalidation to SuperQ".

Once SuperQ receives the invalidation, it will start monitoring particular streams transactions that could be still in SuperQ and wait for them to retire. If there is no such requests, this process would be immediate. The invalidation will be forwarded to L3 as SuperQ ensures there is no more requests in its slots with the matching clientID assignment.

Once L3 cache receives the invalidation, it is guaranteed that none of its requests in the TAG pipeline belongs to OLD marked requests from the stream getting the invalidation. SuperQ already guarantees
this process by waiting for the retirement of the marked requests. TAG pipeline controller will stall the operations and wait for the TAG pipeline to clear than send the indication to STATE Array controller. For the ways that are corresponding to invalidation type, all ways for matching stream type will be updated in one shot. Once invalidation is complete, each L3 bank will send the indication to Pixel Arbiter (GAP). Note that multiple invalidations will be serialized in L3.

GAP will collect all invalidation requests from 4 banks (2 banks for GT1) and make sure all are complete before sending this completion back to corresponding L3 arbiters. L3 arbiters will ungate the ingress FIFOs of the completed invalidation as they complete the last step required.

This is exception part of the Read-Only Stream invalidations where state arbiter needs to stop sending new requests to L3 arbiters and ensure all pre-committed requests already sent to L3 arbiters:

- Stop and discard any prefetch requests that may be ongoing.
- Finish any state requests with the length fields.

Once above conditions are satisfied the “state invalidation” will be forwarded to L3 arbiters and state arbiter will stop sending processing any other state requests while processing the invalidation. The ingress FIFOs will be kept blocked towards the arbiter until L3 arbiters send an indicator (all 4 of them) to state arbiter for the completion of the state invalidation. Note that this is an extra step for L3 arbiters only for the case of state invalidation within Read-Only streams.

Note that state arbiter needs to accumulate any new state requests that falls behind the invalidation event and not process them until the state invalidation complete indicator is seen from GAP. It needs to ensure old state data is cleared before sending the new state requests from clients.

There is a possibility where multiple invalidations could exist for a given stream. This is where L3 arbiters need to coordinate when the invalidation requests would be processed. As L3 arbiter receives a particular invalidation to the time response is received from GAP, there could be yet another invalidation for the same stream. However since the ingress FIFOs are blocked, new invalidation request should not be processed before the prior request is complete. When L3 arbiter receives the completion from GAP, it will start processing the new invalidation as if it was received right at the same cycle.

Same rules apply for state arbiter as well where while the completion for a prior invalidation is pending, there may be another invalidation request from the main command streamer. State arbiter will hold off the execution of the newer invalidation until the completions are seen from the L3 arbiters.

Note that there could be third, forth invalidations while the very invalidation is being processed. All invalidations for a given stream could be collapsed while the prior one is being processed.

**Pipelined Flush for Writes**

Pipelined flush for data cluster writes will be propagated from DC directly to L3 arbiters as their buffers are flushed. L3 arbiters will be getting a flush indicator from each DC independently, there is no point of acting on the first flush indicator, it is easier to wait for both DCs to send their flush request and process them together. Once both flush indicators are seen, L3 arbiter will flush all its ingress FIFOs and block outlet of them being processed. L3 arbiter should not be sending any other requests to SuperQs until a completion is seen.
SuperQ as a response to Write Flush will wait for all its slots to retire, this is to prevent any boundary cases and ensure all writes are retired to L3 (if any). Once emptied, SQ will start accumulating "Flush" requests to the defined sets and ways (defined as data cluster reserved ways) and walk through each entry of the L3. These flush requests are to invalidate and evict any modified lines that may be present in L3s.

After all defined ways are walked with FLUSH requests, SuperQ should wait for empty indicator once again. This is to make sure all evicted data is retired towards GAP.

As GAP receives the flush complete indicator from each bank of L3 (4 for GT2 and 2 for GT1), it will ensure the eviction path is retired towards GAM. Once all done, it will send an indicator to PSD units in each half-slice. Same signal will be received by all L3 arbiters and used to un-block their interfaces towards SuperQ.

Note that SuperQ should not release any credits to L3 arbiter when retiring an internally generated flush request.

Similar to RO invalidations, the status of the write flush can also be tracked via register space.

**Global Invalidation**

Once written a global indicator will be sent to L3 arbiters and state arbiter which will kick-off all invalidations at once. The same register space will collect all completion from L3 arbiters and state arbiter to clear the same bit.

Register bit can be polled by s/w to track to completion of all invalidations.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RW/C</td>
<td>0</td>
<td><strong>L3 Global Invalidations:</strong> Once written it will kick off a global invalidation for L3 both on RO and WR streams. S/W is expected to write logic1 to kick-off the invalidation and H/W will clear the bit once all invalidations are complete. Meanwhile S/W can poll the bit to track the completion of invalidation.</td>
</tr>
</tbody>
</table>

**Shared Local Memory (SLM)**

Shared local memory (aka highly-banked memory) is a portion of L3 which will be dedicated to EUs as a local memory when enabled. The accesses are only possible through data cluster with the destination flag set as SLM. In order to support a highly banked design, 2 of the L3 banks are structured to have 16x4KB portion which could be accessed independently per clock. This part of the L3 can support 16 dw size accesses (per SLM) in a given clock cycle.

These 16 banks can either be used as L3/URB or used as shared local memory with parallel accesses to all banks. The choice of enabling SLM mode is done through MMIO programming.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 0   | RW/C   | 0             | **Enable Shared Local Memory**: When set, it enables the use of 2 banks of L3 as shared local memory which allows 64KB of L3 to be banked as 16x4KB and allows independent accesses to all banks within the same clock cycle. 

*Note: This mode can only be enabled once L3 content is completely flushed.* |

SLM requests are forked around the L3 arbiter, post ingress FIFOs for DC. L3 arbiter delivers request/data to SLM controller upon the availability of credits. Request will be crossed to 2x clock domain routed to corresponding banks. Individual bank controls are managed via SLM controller which are muxed with L3/URB accesses. Note that SLM accesses do carry byte enables and needs to be honored towards the banks. If the request has atomic requirements, SLM controller will provide the data to ALU along with the atomic type. Output data is again managed with SLM controller towards the output cross bars.

SLM should not be accessed through the 3D pipe.

The state control fields are redefined to comprehend L3 addition as follows:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GFDT</td>
<td>GFDT flag to color the displayable surfaces in LLC. This field is later used by GT to poll the LLC during query/flush mechanism and used to push the data to DRAM prior to Display Accesses.</td>
</tr>
</tbody>
</table>
| 1:0  | Cacheability Controls | Control for LLC and L3 cacheability fields  
Bit[1]: LLC cacheability  
Bit[0]: L3 cacheability  
0: Access is NOT cacheable.  
1: Access is cacheable. |

**Dynamic Parity Feature for GFX L3 Cache**

This document is meant to outline and describe dynamic parity detection function in 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 graphics L3 cache, parity detection capability has been added to each sub-bank of the L3. A sub-bank is defined as a 16KB entity of the total 512KB cache. This would translate into 32 independent sub-banks which all have 2 independent redundant rows for a total of 64 possible replacement rows. Redundant rows can be activated to replace a row with identified parity errors via writing the address of the parity error row into hardware registers.

DPF allows the HW to notify SW (driver) when any single bit graphics L3 parity error has occurred and also provide a mechanism to allow SW to fix persistent bit errors where a susceptible bit fails multiple times.
In order to support robust GPGPU / compute workloads on HSW/gen7.5 graphics, Intel recommends that driver developers implement this suggested DPF flow. Regardless, all usages of the graphics L3 cache can benefit from the added parity protection.

**Hardware and Software Flows**

This section contain information on:

- Parity Generation & Detection
- Correction Using Parity Error data and Redundant Rows
- Number of Corrections
- Summary: Basic Algorithm

**Parity Generation & Detection**

The graphics L3 cache will generate 1-bit parity as data is stored in the cache. The parity bit will be written along with data for future verification. As the same content is accessed later in time, HW re-calculates the parity based on read content and compares with the stored parity value.

Once a mismatch is detected, HW generates a parity interrupt for the graphics driver to service. Meanwhile, HW continues forward progress in execution. There is no implicit halt or execution stopping for the HW.

Along with the interrupt, the HW will update a set of registers to indicate which bank/sub-bank/row in which the error has been detected.

**Correction Using Parity Error data and Redundant Rows**

Each sub-bank contains 2 redundant rows which could be used to replace the rows with reported parity errors. The graphics driver, once servicing the parity interrupt, will access the reporting registers and record the bank/sub-bank/row information. This information should be stored in a permanent (non-volatile memory: ie: disk, registry or similar) location for future use by the graphics driver.

The graphics driver will then reset the render engine (i.e. render specific reset) to prevent propagation of the data with the parity error. Even though the graphics driver will terminate the context via resetting the GPU upon a parity interrupt, there is a possibility that the parity interrupt may be observed by the driver after the graphics context is complete or about to switch to a new context. For non-graphics workloads that require high data integrity, such as GPGPU computing, the driver should prevent this possible boundary case by polling the error reporting registers when a context completion or context switch interrupt is registered by the driver. Upon identifying that the error reporting registers are active, the driver should follow the same steps as servicing a parity interrupt.

Post any reset event (graphics reset, power up, etc...), the graphics driver retrieve the parity failure row information from non-volatile storage. It will then program the parity failure row information into corresponding bank/sub-bank registers and start normal graphics operation. Parity failure rows would then be effectively replaced with the extra redundant rows until next system reset.
In the extremely rare probability that redundant rows themselves have a parity errors, the parity error will be reported as the row they have replaced. SW drivers should recognize the use of redundant row and skip the replacement.

**Number of Corrections**

Given DPF is designed to deal with persistent errors, graphics drivers need to be able to identify which sub-bank rows are producing the most number of errors. Hence, the driver should keep a list of the reported parity error rows and record the number of times each row reports a parity error. If there are more than 2 parity error rows identified for a given sub-bank, the driver should replace the top two rows first (decided by historical error count). This will force rows with consistent parity errors to bubble up to the top of the list to be replaced.

SW driver tracking of parity error rows should be saved in non-volatile memory, so the driver can keep track of parity failure rows across reboot/reset.

**Summary**

Basic Algorithm:

- Each sub-bank has two redundant rows. (By HW design)
- Driver SW keeps track of each sub-bank's parity failure rows and keeps a count of failures of each row.
- Error count row information is saved to non-volatile memory, so it persists across reboot and graphics resets.
- At all times, the two redundant sub-bank rows are used to replace the highest count parity failure rows.
- SW always forces reset (graphics render reset is sufficient) on L3 errors.
- SW programs all necessary replacement rows after any reset.

**Sub-banks with more than two persistent parity error rows**

While not expected during normal lifetime operation, a problematic case could occur when the HW reports more than 2 rows on a particular sub-bank are causing parity errors. For this case, the graphics driver should keep replacing the rows, always selecting the two rows with the highest parity error failure count.

**Interrupt Enabling**

In order to enable proper DPF notification, the graphics driver must enable the correct interrupt paths from render command streamer (x20A8) as well as rest of the interrupt structures around GTISR (x44010), GTIMR (x44014), GTIIR (x44018), GTIER (x4401C). Bit5 has been selected for L3 parity interrupts. Please refer to the appropriate register sections for generic graphics interrupt enabling details.
Clearing the Error Reporting Registers

Clearing error status registers for the L3 cache should be done by writing to L3CD Error Status Register bit[13] with a logical value of 1. This should be done after an error is reported and a row has been replaced. The graphics driver can do this in two different ways:

1. Via batch buffer executed in HW (via LRI or LRM mechanisms)
2. Direct writes to graphics HW MMIO space

Direct reads and writes using graphics MMIO (for both error log and status registers) requires DOP level clock gating to be turned off as HW might have finished execution, which will result in a hang when accessing the L3 parity registers. This requires graphics MISCCPCTL bit 0 (x9424[0]) to be cleared prior to register updates. It must be set again post register updates. Do not leave the DOP clock gating bit cleared. Doing so will significantly affect graphics power.

Note that the HW parity registers will clear with any reset, hence error and row replacement registers will have to be re-programmed any time the driver performs a HW reset or driver is re-loaded.

Software Requirement on Silent Data Corruptions

HSW C-step added a mechanism to force h/w to hang while preventing the corrupted data to be exposed outside GPU.

It is expected for s/w to enable silent data corruption prevention for contexts where it matters.

Hardware Registers

Various registers are used to report and contain the parity error failing row information:
Error Report Registers

**L3CDERRST1 - L3CD Error Status Register 1**

<table>
<thead>
<tr>
<th>B/D/F/Type</th>
<th>0/0/0/SARBunit_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address Offset:</strong></td>
<td>B008-B00Bh</td>
</tr>
<tr>
<td><strong>Default Value:</strong></td>
<td>00000000h</td>
</tr>
<tr>
<td><strong>Access:</strong></td>
<td>RW; RO; WO;</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>32 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:25| RO     | 0000000b      | **Reserved (RSVD):**  
Reserved                                                               |
| 24:14| RWC    | 00000000000b  | **Parity row address error (PRTYROWNUM):**  
Data array address which has parity B1:  
Report the data array address which has the Error  
ltcd_sarb_parity_err_rownum[10:0]  
Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13 . |
| 13   | RWC    | 0b            | **Parity Error Valid (PRTYERRVLD):**  
Parity Error valid  
Report the Parity Error  
ltcd_sarb_parity_err_valid  
Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13 . |
| 12:11| RWC    | 00b           | **Parity error bank number (PRTYBNKNUM):**  
bank number which has parity error  
Report the bank no. which has the Error  
ltcd_sarb_parity_err_banknum[1:0]  
Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13 . |
<p>| 10:8 | RWC    | 000b          | <strong>Parity Error sub-bank no (PRTYSBNKNUM):</strong>                                      |</p>
<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Parity Error in sub bank: ltcd0_sarb_parity_err_subanknum[2:0] Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>1b</td>
<td>Core</td>
<td><strong>Parity report enable (LCPRTYRPTEN):</strong> sarbcf_csr_lc_parity_report_en this is the parity reporting enable, by default it is enabled. When enabled parity will be reported by ltcd to sarb. When disabled by driver, ltcd should not send out any parity error to SARB.</td>
</tr>
<tr>
<td>6:0</td>
<td>RO</td>
<td>00h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>
Row Replacement Registers

The range of row replacement registers is addresses xB070 (bank0/sub-bank0) to xB0EC (bank3/sub-bank7). Only the first register format is given below - all registers have the same format.

L3B0REG0 - L3 bank0 reg0 log error

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>RW</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error1 (RNUMERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error.</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>16</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 1 (VLDERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>RW</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error0 (RNUMERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error.</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>Bits</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong>&lt;br&gt;Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3 Register Space**

Note that the L3 register space is not cleared in engine or device specific resets. The registers in question need to be reprogrammed completely to known values at the context creation time.

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.)
**SARERRST0 - SARB Error Status slice0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B004-B007h
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

Reports the error if any has occurred for certain sarb features.

This register is not ctx saved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31  | RO     | 0b            | Core    | **Error if general bound is zero (ERRGENBDZO):**  
|     |        |               |         | Error if general bound is zero set by sarbunit  
|     |        |               |         | 1: general bound address is 0  
|     |        |               |         | sarbcf_csr_gen_bnd_zero_err |
| 30  | RO     | 0b            | Core    | **Error if dynamic bound is zero (ERRDYDNZO):**  
|     |        |               |         | Error if dynamic bound is zero set by sarbunit  
|     |        |               |         | 0: no error  
|     |        |               |         | 1: dynamic address is 0  
|     |        |               |         | sarbcf_csr_dyn_bnd_zero_err |
| 29  | RO     | 0b            | Core    | **Reserved (RSVD):** |
| 28  | RO     | 0b            | Core    | **General Bound Check Overflow Error (GENBNDOVF):**  
|     |        |               |         | General Bound Check Overflow Error - set by sarbunit  
|     |        |               |         | 1: overflow for general bound check  
|     |        |               |         | sarbcf_csr_gen_bnd_ovflw_err |
| 27  | RO     | 0b            | Core    | **Dynamic Bound Check Overflow Error (DYNBDOVF):**  
|     |        |               |         | Dynamic Bound Check Overflow Error - set by sarbunit  
|     |        |               |         | 1: overflow for dynamic bound check  
<p>|     |        |               |         | sarbcf_csr_dyn_bnd_ovflw_err |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 26  | RO     | 0b            | Core    | **Lower Bound Check Overflow Error (LWRBDOVF):**  
|     |        |               |         | Lower Bound Check Overflow Error-set by sarbunit  
|     |        |               |         | lower bound overflow  
|     |        |               |         | sarbcf_csr_lower_bnd_err |
| 25:21 | RO     | 00000b        | Core    | **INVALIDATION FLUSH STATUS REPORTING (INVSTRPT):**  
|      |        |               |         | invalidation status for l3 is reported in this register. |
| 20:18 | RO     | 000b          | Core    | **SARB invalidation Status reporting (SARBINVSTRPT):**  
|      |        |               |         | invalidation status of sarb is reported in this register. |
| 17  | RO     | 0b            | Core    | **HW surface Bound Check Overflow Error (HWSBDOVF):**  
|     |        |               |         | sarbcf_csr_hw_surf_bnd_ovflw_err  
|     |        |               |         | HW Surface Bound Check Overflow Error -set by sarbunit  
|     |        |               |         | 1: overflow for bound check |
| 16  | RO     | 0b            | Core    | **Error if hw surface bound is zero (ERRHWSNZO):**  
|     |        |               |         | sarbcf_csr_hw_surf_bnd_zero_err  
|     |        |               |         | Error if hw surface bound is zero- set by sarbunit  
|     |        |               |         | 0:no error  
|     |        |               |         | 1: address is 0 |
| 15  | RO     | 0b            | Core    | **buffer Ready intp err (INTPERR):**  
|     |        |               |         | When both buffers are ready before one buffer ready is cleared by sft sarb will generate intp err (it is not expected that second buffer ready should assert while first buffer ready was not cleared by sftwr.)  
|     |        |               |         | sarbcf_both_buffer_rd_intp_err |
| 14:0 | RO     | 0000h         | Core    | **Reserved (RSVD):**  
|      |        |               |         | Reserved |
L3CDERRST01 - L3CD Error Status register 1 slice 0

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B008-B00Bh

Default Value: 00000080h

Access: RW; RO; WO;

Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>RO</td>
<td>0000000b</td>
<td>Core</td>
<td>Reserved (RSVD): Reserved</td>
</tr>
<tr>
<td>24:14</td>
<td>RWC</td>
<td>00000000000b</td>
<td>Core</td>
<td>Parity row address error (PRTYROWNUM): Data array address which has parity B1: Report the data array address which has the Error ltcd_sarb_parity_err_rownum[10:0] Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>13</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>Parity Error Valid (PRTYERRVLD): Parity Error valid Report the Parity Error ltcd_sarb_parity_err_valid Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>12:11</td>
<td>RWC</td>
<td>00b</td>
<td>Core</td>
<td>Parity error bank number (PRTYBNKNUM): bank number which has parity error Report the bank no. which has the Error ltcd_sarb_parity_err_banknum[1:0] Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 10:8 | RWC    | 000b          | Core    | **Parity Error sub-bank no (PRTYSBNKNUM):**  
Parity Error in sub bank:  
ltcd0_sarb_parity_err_subanknum[2:0]  
Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13. |
| 7   | RW     | 1b            | Core    | **Parity report enable (LCPRTYRPTEN):**  
sarbcf_csr_lc_parity_report_en  
this is the parity reporting enable, by default it is enabled.  
When enabled by driver parity will be reported by ltcd to sarb.  
When disabled by driver, ltcd should not send out any parity error to SARB.  
this register bit is used by both slices |
| 6:0 | RO     | 00h           | Core    | **Reserved (RSVD):** |
### L3CDERRST02 - L3CD Error Status register 2 slice 0

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B00C-B00Fh
- **Default Value:** 00000000h
- **Access:** RO; RWC;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:29</td>
<td>RO</td>
<td>000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong> reserved</td>
</tr>
<tr>
<td>28</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>URB High Limit Error on B3 (URBHLB3):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>URB High Limit Error on B3:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Report the URB High Limit Error- Address Bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Once set, it can be cleared only by MMIO Write to this register. A write of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>value 1 will clear it</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the MMIO as Error status. This can be only cleared by MMIO Write to that</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltcc3_sarb_urboff_error</td>
</tr>
<tr>
<td>27</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>URB High Limit Error on B2 (URBHLB2):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>URB High Limit Error on B2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Report the URB High Limit Error- Address Bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Once set, it can be cleared only by MMIO Write to this register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the MMIO as Error status. This can be only cleared by MMIO Write to that</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltcc2_sarb_urboff_error</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 26  | RWC    | 0b            | Core    | **URB High Limit Error on B1 (URBHLB1):**  
URB High Limit Error on B1:  
Report the URB High Limit Error - Address Bound check  
Once set, it can be cleared only by MMIO Write to this register.  
( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in the MMIO as Error status. This can be only cleared by MMIO Write to that Bit. )  
ltcc1_sarb_urboff_error |
| 25  | RWC    | 0b            | Core    | **URB High Limit Error on B0 (URBHLB0):**  
URB High Limit Error on B0:  
Report the URB High Limit Error - Address Bound check  
Once set, it can be cleared only by MMIO Write to this register.  
( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in the MMIO as Error status. This can be only cleared by MMIO Write to that Bit. )  
ltcc0_sarb_urboff_error |
| 24:0| RO     | 0000000h      | Core    | **Reserved (RSVD):** |
## L3SQCREG1 - L3 SQC registers 1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B010-B013h

**Default Value:** 01610000h

**Access:** RW; RO;

**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:28 | RO     | 0000b         | Core    | **Reserved (RSVD):**  
|       |        |               |         | Reserved     |
| 27    | RW     | 0b            | Core    | **Convert L3 cycle for texture to uncachable (CON4TXTUNC):**  
|       |        |               |         | Convert L3 cycle for texture to uncachable  
|       |        |               |         | 1: texture has no way allocation in L3  
|       |        |               |         | 0: texture has atleast 1 way allocated in L3 (default)  
|       |        |               |         | sarbcf_csr_lsqc_cnvt_txt_unchble |
| 26    | RW     | 0b            | Core    | **Convert L3 cycle for constant to uncachable (CON4CONSUNC):**  
|       |        |               |         | Convert L3 cycle for constant to uncachable  
|       |        |               |         | 1: constant has no way allocation in L3  
|       |        |               |         | 0: constant has atleast 1 way allocated in L3 (default)  
|       |        |               |         | sarbcf_csr_lsqc_cnvt_const_unchble |
| 25    | RW     | 0b            | Core    | **Convert L3 cycle for Inst/State to uncachable (CON4INSSTUNC):**  
|       |        |               |         | Convert L3 cycle for Inst/State to uncachable  
|       |        |               |         | 1: Inst/State has no way allocation in L3  
|       |        |               |         | 0: Inst/State has atleast 1 way allocated in L3 (default)  
<p>|       |        |               |         | sarbcf_csr_lsqc_cnvt_ins_st_unchble |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>RW</td>
<td>1b</td>
<td>Core</td>
<td><strong>Convert L3 cycle for DC to uncachable (CON4DCUNC):</strong>&lt;br&gt;Convert L3 cycle for DC to uncachable&lt;br&gt;1: DC has no way allocation in L3 (default)&lt;br&gt;0: DC has atleast 1 way allocated in L3&lt;br&gt;sarbcf_csr_lsqc_cnvt_dc_unchble&lt;br&gt;&lt;em&gt;Note: This bit can not be set to &quot;1&quot; when atomics in L3 mode is enabled&lt;/em&gt;</td>
</tr>
<tr>
<td>23:19</td>
<td>RW</td>
<td>01100b</td>
<td>Core</td>
<td><strong>L3SQ General Priority Credit Initialization (SQGPCI):</strong>&lt;br&gt;L3SQ General Priority Credit Initialization (SQGPCI):&lt;br&gt;Number of general and high priority credits that SQ presents to L3 Arbiter blocks. This inherently also determines the depth of the SQ; reduce the number of credits and SQ will use fewer slots.&lt;br&gt;Any value not listed here, is considered Reserved.&lt;br&gt;gen priority credits is always greater than high priority credits&lt;br&gt;&lt;br&gt;&lt;strong&gt;Value&lt;/strong&gt;</td>
</tr>
<tr>
<td>00000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>24(default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other values are not possible</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Need to go upto 32 credits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sarbcf_csr_lsqc_gen_credit_init[4:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>18:14</td>
<td>RW</td>
<td>00100b</td>
<td>Core</td>
<td>L3SQ High Priority Credit Initialization (SQHPCI):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of general and high priority credits that SQ presents to L3 Arbiter blocks. This inherently also determines the depth of the SQ; reduce the number of credits and SQ will use fewer slots.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Any value not listed here, is considered Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>gen priority credits is always greater than high priority credits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00011</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Other values are not possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_hp_credit_init[4:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_hp_credit_init[4:0] ++</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_gen_credit_init[4:0] should always be less than or equal to 32.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>13:12</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>L3SQ Atomics Credit Initialization (SQACI):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Atomics Credit Initialization (SQACI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of atomics credits that SQ presents to L3 Arbiter blocks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = 2 Credits (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = 1 Credit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1X = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_atom_credit_init[1:0]</td>
</tr>
<tr>
<td>11:10</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>L3SQ Data Credit Initialization (SQDCI):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Data Credit Initialization (SQDCI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of data credits that SQ presents to L3 Arbiter blocks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = 4 Credits (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = 1 Credit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = 2 Credits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = 3 Credits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_data_credit_init[1:0]</td>
</tr>
<tr>
<td>9</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>L3SQ Read Once Enable for Sampler Client (SQROE):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Read Once Enable for Sampler Client (SQROE):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enables Read Once indications to L3 Cache from SQ. Once enabled, any reads</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>from Sampler client (MT) will be sent as Read Once</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Reads from Sampler clients issue Read to L3 Cache (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Reads from Sampler clients issue Read Once to L3 Cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_sampler_readonce_en</td>
</tr>
</tbody>
</table>
### L3SQ Outstanding GAP Reads (SQOUTSGAP):

Identifies the number of Pixel Arbiter Reads that can be outstanding before SQ throttles the puts to GAP. This is not an exact limit, but instead it is used as a threshold to throttling; once the read count is greater than or equal to the threshold, then no reads will be issued until data returns are received to bring the outstanding count back below the threshold.

- **000** = No limit (default)
- **001** = 2 reads
- **010** = 3 reads
- **011** = 5 reads
- **100** = 9 reads
- **101** = 17 reads
- **11X** = Reserved

```
sarbcf_csr_lsqc_outs_gaprd[2:0]
```

### L3SQ Outstanding L3 Fills (SQOUTSL3F):

Identifies the number of L3 Fills that can be outstanding before SQ throttles the fill requests to L3 Cache. This is not an exact limit, but instead it is used as a threshold to throttling; once the fill count is greater than or equal to the threshold, then no fills will be issued until the fill responses are received to bring the outstanding count back below the threshold.

- **000** = No limit (default)
- **001** = 1 fill
- **010** = 2 fills
- **011** = 4 fills
- **100** = 8 fills
- **101** = 16 fills
- **11X** = Reserved

```
sarbcf_csr_lsqc_outs_fill[2:0]
```
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2:0 | RW     | 000b          | Core    | **L3SQ Outstanding L3 Lookups (SQOUTSL3L):**  
Identifies the number of L3 lookups that can be outstanding before SQ throttles the lookup requests to L3 Cache. This is not an exact limit, but instead it is used as a threshold to throttling; once the lookup count is greater than or equal to the threshold, then no lookups will be issued until the lookup responses are received to bring the outstanding count back below the threshold.  
000 = No limit (default)  
001 = 1 lookup  
010 = 2 lookups  
011 = 4 lookups  
100 = 8 lookups  
101 = 16 lookups  
11X = Reserved  
sarbcf_csr_lsqc_outs_lookup[2:0] |
### L3SQCREG2 - L3 SQC registers 2

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B014-B017h

**Default Value:** 00004567h

**Access:** RO; RW;

**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:17</td>
<td>RO</td>
<td>000000000000000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong> Reserved</td>
</tr>
<tr>
<td>16</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>L3SQ Priority Selection Disable (SQPRIDIS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Priority Selection Disable (SQPRIDIS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enables the use of priority selection based on client ID decodes. If</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>disabled, all cycles in SQ will be treated as same priority.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Priority selection is enabled (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Priority selection is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>sarbcf_csr_priority_cnt_disable</code></td>
</tr>
<tr>
<td>15</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>L3SQ Priority 3 Pool Count Disable (SQPRI3CNTDIS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Priority 3 Pool Count Disable (SQPRI3CNTDIS):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When set, priority3 pool becomes unlimited. And priority3 pool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>count value should not be used in reset of the remaining counters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Priority 3 pool count is enabled (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Priority 3 pool count is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>sarbcf_csr_priority3_cnt_disable</code></td>
</tr>
<tr>
<td>14:12</td>
<td>RW</td>
<td>100b</td>
<td>Core</td>
<td><strong>L3SQ Priority 3 Pool Counter (SQPRI3CNT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Priority 3 Pool Counter (SQPRI3CNT):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The count of cycles will be selected from priority3 pool before</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>switching to other priority pools. Count is used as the power of 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000 = 1 request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>001 = 2 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>010 = 4 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>011 = 8 requests</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>----------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>L3SQ Priority 2 Pool Count Disable (SQPRI2CNTDIS):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:8</td>
<td>RW</td>
<td>101b</td>
<td>Core</td>
<td>L3SQ Priority 2 Pool Counter (SQPRI2CNT):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>L3SQ Priority 1 Pool Count Disable (SQPRI1CNTDIS):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>RW</td>
<td>110b</td>
<td>Core</td>
<td>L3SQ Priority 1 Pool Counter (SQPRI1CNT):</td>
</tr>
</tbody>
</table>

111 = 128 requests
sarbcf_csr_priority3_cnt[2:0]

L3SQ Priority 2 Pool Count Disable (SQPRI2CNTDIS):
L3SQ Priority 2 Pool Count Disable (SQPRI2CNTDIS):
When set, priority2 pool becomes unlimited. And priority2 pool count value should not be used in reset of the remaining counters.
0 = Priority 2 pool count is enabled (default)
1 = Priority 2 pool count is disabled
sarbcf_csr_priority2_cnt_disable

L3SQ Priority 2 Pool Counter (SQPRI2CNT):
L3SQ Priority 2 Pool Counter (SQPRI2CNT):
The count of cycles will be selected from priority2 pool before switching to other priority pools. Count is used as the power of 2.
000 = 1 request
001 = 2 requests
010 = 4 requests
011 = 8 requests
111 = 128 requests
sarbcf_csr_priority2_cnt[2:0]

L3SQ Priority 1 Pool Count Disable (SQPRI1CNTDIS):
L3SQ Priority 1 Pool Count Disable (SQPRI1CNTDIS):
When set, priority1 pool becomes unlimited. And priority1 pool count value should not be used in reset of the remaining counters.
0 = Priority 1 pool count is enabled (default)
1 = Priority 1 pool count is disabled
sarbcf_csr_priority1_cnt_disable

L3SQ Priority 1 Pool Counter (SQPRI1CNT):
L3SQ Priority 1 Pool Counter (SQPRI1CNT):
The count of cycles will be selected from priority1 pool before
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>switching to other priority pools. Count is used as the power of 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000 = 1 request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>001 = 2 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>010 = 4 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>011 = 8 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111 = 128 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_priority1_cnt[2:0]</td>
</tr>
</tbody>
</table>

**3** | RW | 0b | Core | **L3SQ Priority 0 Pool Count Disable (SQPRI0CNTDIS):** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Priority 0 Pool Count Disable (SQPRI0CNTDIS):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When set, priority0 pool becomes unlimited. And priority0 pool count value should not be used in reset of the remaining counters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Priority 0 pool count is enabled (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Priority 0 pool count is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_priority0_cnt_disable</td>
</tr>
</tbody>
</table>

**2:0** | RW | 111b | Core | **L3SQ Priority 0 Pool Counter (SQPRI0CNT):** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3SQ Priority 0 Pool Counter (SQPRI0CNT):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The count of cycles will be selected from priority0 pool before switching to other priority pools. Count is used as the power of 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000 = 1 request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>001 = 2 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>010 = 4 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>011 = 8 requests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111 = 128 requests (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_priority0_cnt[2:0]</td>
</tr>
</tbody>
</table>

**L3SQCREG3 - L3 SQC registers 3**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B018-B01Bh

**Default Value:** 00001ABFh
## Bit Access Bit Access Default Value Default Value RST/PWR RST/PWR Description Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:30</td>
<td>RO</td>
<td>00b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong>&lt;br&gt;Reserved</td>
</tr>
<tr>
<td>29:28</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>SOLunit Priority Value (SQSOLPRIVAL):</strong>&lt;br&gt;SOLunit Priority Value (SQSOLPRIVAL):&lt;br&gt;Identifies the priority value for all cycles that are initiated by SOLunit. Priority is used in the L3 Super Queue (L3SQ).&lt;br&gt;00 = Priority 0 (default)&lt;br&gt;01 = Priority 1&lt;br&gt;10 = Priority 2&lt;br&gt;11 = Priority 3&lt;br&gt;sarbcf_csr_sol_priority[1:0]</td>
</tr>
<tr>
<td>27:26</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>GSunit Priority Value (SQGSPRIVAL):</strong>&lt;br&gt;GSunit Priority Value (SQGSPRIVAL):&lt;br&gt;Identifies the priority value for all cycles that are initiated by GSunit. Priority is used in the L3 Super Queue (L3SQ).&lt;br&gt;00 = Priority 0 (default)&lt;br&gt;01 = Priority 1&lt;br&gt;10 = Priority 2&lt;br&gt;11 = Priority 3&lt;br&gt;sarbcf_csr_gs_priority[1:0]</td>
</tr>
<tr>
<td>25:24</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>TEunit Priority Value (SQTEPRIVAL):</strong>&lt;br&gt;TEunit Priority Value (SQTEPRIVAL):&lt;br&gt;Identifies the priority value for all cycles that are initiated by TEunit. Priority is used in the L3 Super Queue (L3SQ).&lt;br&gt;00 = Priority 0 (default)&lt;br&gt;01 = Priority 1&lt;br&gt;10 = Priority 2&lt;br&gt;11 = Priority 3</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>23:22</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>CLunit Priority Value (SQCLPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by CLunit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_te_priority[1:0]</td>
</tr>
<tr>
<td>21:20</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>TSunit Priority Value (SQTSPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by TSunit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_ts_priority[1:0]</td>
</tr>
<tr>
<td>19:18</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>SFunit Priority Value (SQSFPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by SFunit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_sf_priority[1:0]</td>
</tr>
<tr>
<td>17:16</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>SVSM Priority Value (SQSVSPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SVSM Priority Value (SQSVSPRIVAL):</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by SVSM. Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_svsm_priority[1:0]</td>
</tr>
<tr>
<td>15:14</td>
<td>RW</td>
<td>00b</td>
<td>Core</td>
<td><strong>SARB Priority Value (SQSARBPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SARB Priority Value (SQSARBPRIVAL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by State Arbiter (SARB). Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_sarb_priority[1:0]</td>
</tr>
<tr>
<td>13:12</td>
<td>RW</td>
<td>01b</td>
<td>Core</td>
<td><strong>SBE Priority Value (SQSBEPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SBE Priority Value (SQSBEPRIVAL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by SBE. Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_sbe_priority[1:0]</td>
</tr>
<tr>
<td>11:10</td>
<td>RW</td>
<td>10b</td>
<td>Core</td>
<td><strong>IC$ Priority Value (SQICPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IC$ Priority Value (SQICPRIVAL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by Instruction Cache (IC$). Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|     |        | 10b           | Core    | 10 = Priority 2 (default)  
|     |        |               |         | 11 = Priority 3  
|     |        |               |         | sarbcf_csr_ic_priority[1:0] |
| 9:8 | RW     | 10b           | Core    | **TDL Priority Value (SQTDLPRIVAL):**  
|     |        |               |         | TDL Priority Value (SQTDLPRIVAL):  
|     |        |               |         | Identifies the priority value for all cycles that are initiated by TDL. Priority is used in the L3 Super Queue (L3SQ).  
|     |        |               |         | 00 = Priority 0  
|     |        |               |         | 01 = Priority 1  
|     |        |               |         | 10 = Priority 2 (default)  
|     |        |               |         | 11 = Priority 3  
|     |        |               |         | sarbcf_csr_tdl_priority[1:0] |
| 7:6 | RW     | 10b           | Core    | **DCunit Priority Value (SQDCPRIVAL):**  
|     |        |               |         | DCunit Priority Value (SQDCPRIVAL):  
|     |        |               |         | Identifies the priority value for all cycles that are initiated by DC. Priority is used in the L3 Super Queue (L3SQ).  
|     |        |               |         | 00 = Priority 0  
|     |        |               |         | 01 = Priority 1  
|     |        |               |         | 10 = Priority 2 (default)  
|     |        |               |         | 11 = Priority 3  
|     |        |               |         | sarbcf_csr_dc_priority[1:0] |
| 5:4 | RW     | 11b           | Core    | **DAPR Priority Value (SQDAPRPRIVAL):**  
|     |        |               |         | DAPR Priority Value (SQDAPRPRIVAL):  
|     |        |               |         | Identifies the priority value for all cycles that are initiated by DAPR. Priority is used in the L3 Super Queue (L3SQ).  
|     |        |               |         | 00 = Priority 0  
|     |        |               |         | 01 = Priority 1  
|     |        |               |         | 10 = Priority 2  
|     |        |               |         | 11 = Priority 3 (default)  
<p>|     |        |               |         | sarbcf_csr_dapr_priority[1:0] |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:2</td>
<td>RW</td>
<td>11b</td>
<td>Core</td>
<td><strong>MTunit Priority Value (SQMTPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>MTunit Priority Value (SQMTPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by Sampler</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(MT). Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_mt_priority[1:0]</td>
</tr>
<tr>
<td>1:0</td>
<td>RW</td>
<td>11b</td>
<td>Core</td>
<td><strong>LSQCunit Priority Value (SQPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>LSQCunit Priority Value (SQPRIVAL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identifies the priority value for all cycles that are initiated by Super</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Queue (L3 Evictions). Priority is used in the L3 Super Queue (L3SQ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 = Priority 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 = Priority 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 = Priority 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 = Priority 3 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lsqc_priority[1:0]</td>
</tr>
</tbody>
</table>

**L3CNTLREG1 - L3 Control Register1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B01C-B01Fh

**Default Value:** 8C47FF80h

**Access:** RW; RO;

**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:28</td>
<td>RW</td>
<td>1000b</td>
<td>Core</td>
<td><strong>Data Fifo Depth Control (DFIFODC):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Data Fifo Depth Control (DFIFODC):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stall Control: POR is 1000b. Flexing for Hitting the stall Validation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>scenarios</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Maximum available setting for h/w is &quot;1000&quot;, any higher setting will not be functional.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_datafifo_depth[3:0]</td>
</tr>
<tr>
<td>27:24</td>
<td>RW</td>
<td>1100b</td>
<td>Core</td>
<td><strong>Data Clock off time (DCLKOFFT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Clock off time (DATACLKOFF):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Clock off time - Data block is shut off after these many number of clocks programmed in this register bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_dataclkoff_time[3:0]</td>
</tr>
<tr>
<td>23:20</td>
<td>RW</td>
<td>0100b</td>
<td>Core</td>
<td><strong>TAG CLK OFF TIME (TAGCLKOFF):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TAG CLK OFF TIME (TAGCLKOFF):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TAG Clock Off time. This is the time, which Clock gating Logic check before it turn off the clock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_tagclkoff_time[3:0]</td>
</tr>
<tr>
<td>19</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>L3 Aging Disable Bit (L3AGDIS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3 Aging Disable Bit (L3AGDIS):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aging Disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_agingdis</td>
</tr>
<tr>
<td>18:15</td>
<td>RW</td>
<td>1111b</td>
<td>Core</td>
<td><strong>Fill aging (L3AGF):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fill aging (L3AGF):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aging Counter for Fill</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_fill_aging_cnt[3:0]</td>
</tr>
<tr>
<td>14:11</td>
<td>RW</td>
<td>1111b</td>
<td>Core</td>
<td><strong>Aging Counter for Read 1 Port (L3AGR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aging Counter for Read 1 Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_rd1_aging_cnt[3:0]</td>
</tr>
<tr>
<td>10:7</td>
<td>RW</td>
<td>1111b</td>
<td>Core</td>
<td><strong>L3 Aging Counter for R0 (L3AGR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3 Aging Counter for R0 (L3AGR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aging Counter for R0 Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lc_rd0_aging_cnt[3:0]</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 6:3   | RW     | 0000b         | Core    | **Number of NOPs (L3NOP):**  
Number of NOPs (L3NOP):  
Number of NOPs to be inserted between the Tag commands.  
sarbcf_csr_lc_num_nop[3:0] |
| 2     | RW     | 0b            | Core    | **OP0/OP1 Disable (L3OPDIS):**  
OP0/OP1 Disable (L3OPDIS):  
This bit is used to enable the feature of inserting the number of cycles  
between the tag pipeline operation.  
sarbcf_csr_lc_op0op1_disable |
| 1     | RW     | 0b            | Core    | **L3 OP1 Disable Mode (L3OP1DIS):**  
L3 OP1 Disable Mode (L3OP1DIS):  
OP1 in L3 can be disabled which means there will be one Command  
transferred to the Tag pipeline in 1X Domain  
sarbcf_csr_lc_op1_disable  
Note: If this bit is set Aging mode needs to be disabled as well. |
| 0     | RO     | 0b            | Core    | **Reserved (RSVD):**  
Reserved |

**L3CNTLREG2 - L3 Control Register2**

The following register has GT2 sizes as given and GT1 sizes in parenthesis.  
GT3 sizes will be always 2X the GT2 sizes listed (i.e. if 000001: 8KB (4KB for GT1) for GT3 it will be 16KB).

<table>
<thead>
<tr>
<th>B/D/F/Type:</th>
<th>0/0/0/SARBunit_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Offset:</td>
<td>B020-B023h</td>
</tr>
<tr>
<td>Default Value:</td>
<td>00080040h</td>
</tr>
<tr>
<td>Access:</td>
<td>RW; RO;</td>
</tr>
<tr>
<td>Size:</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>31:28</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>27</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>DC way assignment SLM Behavior (DCWASLMB):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DC way assignment SLM Behavior: In shared local memory mode DC is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>assigned into low b/w mode which requires it to be assigned non-matched</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ways of bigger banks and it will be hashed to 2 banks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_dc_slm_lowbw</td>
</tr>
<tr>
<td>26:21</td>
<td>RW</td>
<td>000000b</td>
<td>Core</td>
<td><strong>DC Way Assignment (DCWASS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DC Way Assignment: Number of ways allocated for DC. Note this allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>is only for DC data types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000: 0KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000001: 8KB (4KB for GT1, 16KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000010: 16KB (8KB for GT1, 32KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111111: 504KB (252KB for GT1, 1008KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: This field must be 0KB is All L3 Client Pool is non-zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_dc_size[5:0]</td>
</tr>
<tr>
<td>20</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>RO Client Pool SLM Behavior (ROCPSSLMB):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RO Client Pool SLM Behavior: In shared local memory mode Read Only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clients are assigned into low b/w mode which requires it to be assigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>non-matched ways of bigger banks and it will be hashed to 2 banks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_ro_slm_lowbw</td>
</tr>
<tr>
<td>19:14</td>
<td>RW</td>
<td>100000b</td>
<td>Core</td>
<td><strong>Read Only Client Pool (RDOCPL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read Only Client Pool: Number of ways allocated for ROnly L3 clients.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This is a combined pool for all RO clients.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In GT2 it is represented interms of 8KB and GT1 it is represented interms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of 4KB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000: 0KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000001: 8KB (4KB for GT1, 16KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000010: 16KB (8KB for GT1, 32KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111111: 504KB (252KB for GT1, 1008KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: If all ROCclient pool is non-zero, than Inst/state, Const and Texture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>client allocation should have 0KB allocation.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_ro_size[5:0]</td>
</tr>
<tr>
<td>13:8</td>
<td>RO</td>
<td>0000000b</td>
<td>Core</td>
<td><strong>RSVD (RSVD):</strong></td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>URB SLM Behavior (URBSLMB):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>URB SLM Behavior: In shared local memory mode URB is assigned into low b/w mode which requires it to be assigned non-matched ways of bigger banks and it will be hashed to 2 banks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_urb_slm_lowbw</td>
</tr>
<tr>
<td>6:1</td>
<td>RW</td>
<td>100000b</td>
<td>Core</td>
<td><strong>URB Allocation (URBALL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>URB Allocation: Number of ways allocated for URB usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000: 0KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000001: 8KB (4KB for GT1, 16KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000010: 16KB (8KB for GT1, 32KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111111: 504KB (252KB for GT1, 1008 KB for GT3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_urb_size[5:0]</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>SLM Mode Enable (SLMMENB):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLM Mode Enable: When enabled, a 128KB region of L3 is reserved for SLM. This allocation is done on 2 banks with 64KB per half-slice.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: SLM is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: SLM is enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: For GT1, there is only one 64KB allocation for single half-slice.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_slm_mode</td>
</tr>
</tbody>
</table>

**L3CNTLREG3 - L3 Control Register3**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B024-B027h

**Default Value:** 00000000h

**Access:** RO; RW;

**Size:** 32 bits
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:22</td>
<td>RO</td>
<td>0000000000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
</tbody>
</table>
| 21   | RW     | 0b            | Core    | **Textures Way Allocation SLM Behavior (TWALSLMB):**  
|      |        |               |         | Textures Way Allocation SLM Behavior: In shared local memory mode Textures is assigned into low b/w mode which requires it to be assigned non-matched ways of bigger banks and it will be hashed to 2 banks.  
sarbcf_csr_tex_slm_lowbw |
| 20:15| RW     | 000000b       | Core    | **Textures Way Allocation (TXWYALL):**  
|      |        |               |         | Textures Way Allocation: Number of ways allocated for Textures. In GT2 it is represented in terms of 8KB and GT1 it is represented in terms of 4KB  
|      |        |               |         | 000000: 0KB  
|      |        |               |         | 000001: 8KB (4KB for GT1, 16KB for GT3)  
|      |        |               |         | 000010: 16KB (8KB for GT1, 32KB for GT3)  
|      |        |               |         | 111111: 504KB (252KB for GT1, 1004KB for GT3)  
|      |        |               |         | Note: This field must be 0KB if All L3 Client Pool or Read-Only Client Pool is non-zero.  
sarbcf_csr_tex_size[5:0] |
| 14   | RW     | 0b            | Core    | **Constants Way Allocation SLM Behavior (CWASLMB):**  
|      |        |               |         | Constants Way Allocation SLM Behavior: In shared local memory mode Instruction/state is assigned into low b/w mode which requires it to be assigned non-matched ways of bigger banks and it will be hashed to 2 banks.  
sarbcf_csr_const_slm_lowbw |
| 13:8 | RW     | 000000b       | Core    | **Constants Way Allocation (CTWALL):**  
|      |        |               |         | Constants Way Allocation: Number of ways allocated for Constants. In GT2 it is represented in terms of 8KB and GT1 it is represented in terms of 4KB  
|      |        |               |         | 000000: 0KB  
|      |        |               |         | 000001: 8KB (4KB for GT1, 16KB for GT3)  
|      |        |               |         | 000010: 16KB (8KB for GT1, 32KB for GT3)  
<p>|      |        |               |         | 111111: 504KB (252KB for GT1, 1004KB for GT3) |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: This field must be 0KB is All L3 Client Pool or Read-Only Client Pool is non-zero. sarbcf_csr_const_size[5:0]</td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>Instruction/State Way Allocation SLM Behavior (ISWASLMB): Instruction/State Way Allocation SLM Behavior: In shared local memory mode Instruction/state is assigned into low b/w mode which requires it to be assigned non-matched ways of bigger banks and it will be hashed to 2 banks. sarbcf_csr_is_slm_lowbw</td>
</tr>
<tr>
<td>6:1</td>
<td>RW</td>
<td>000000b</td>
<td>Core</td>
<td>Instruction/State Way Allocation (ISWYALL): Instruction/State Way Allocation: Number of ways allocated for Instruction/State usage In GT2 it is represented in terms of 8KB and GT1 it is represented in terms of 4KB 000000: 0KB 000001: 8KB (4KB for GT1, 16KB for GT3) 000010: 16KB (8KB for GT1, 32KB for GT3) 111111: 504KB (252KB for GT1, 1008KB for GT3) Note: This field must be 0KB is All L3 Client Pool or Read-Only Client Pool is non-zero. sarbcf_csr_is_size[5:0]</td>
</tr>
<tr>
<td>0</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td>Reserved (RSVD): Reserved</td>
</tr>
</tbody>
</table>

**L3SLMREG - L3 SLM Register**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B028-B02Bh

**Default Value:** 40000000h

**Access:** RO; RW;

**Size:** 32 bits
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>Disable Periodic SLM/SQ slot allocation (DPSLMALL): Disable Periodic SLM/SQ slot allocation: When cfg_lslm_livelock_fairarb_dis=1 lslm unit will always have the higher priority and lslm_lsqc_block to lsqcunit is asserted as long as there are requests in SLM FIFO sarbcf_csr_lslm_livelock_fairarb_dis</td>
</tr>
<tr>
<td>30:26</td>
<td>RW</td>
<td>10000b</td>
<td>Core</td>
<td>LSLM_SQ_PENDING_MAX (LSLMSQPEND): If lslmunit has read data to be sent to lcbunit this cfg register specifies the maximum number of clocks for which LSLMunit can block SQ request from being sent to lcbunit Default value = 8 sarbcf_csr_lslm_sqpend_max[4:0]</td>
</tr>
<tr>
<td>25</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>LSLM address disable (LSLMADDIS): 0- Enable b2b addr maching fix. lslmunit shouldnt block the cycle in fifo if there is a match in the pipeline 1- Disable b2b addr maching fix. lslmunit should block the cycle in fifo if there is a match in the pipeline sarbcf_csr_lslm_same_addr_dis default =0</td>
</tr>
<tr>
<td>24:0</td>
<td>RO</td>
<td>0000000h</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
</tbody>
</table>

**GARBCNTLREG - Arbiter Control Register**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B02C-B02Fh
- **Default Value:** 29124500h
- **Access:** RW, RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>30</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Disables hashing function (DISHHF):</strong>&lt;br&gt;Disables hashing function to generate bank_id[1:0] for L3$ bank accessing, and forces the use of address[7:6] for bank_id[1:0].&lt;br&gt;0 : (default) Hash function enabled to generate L3$ bank IDs.&lt;br&gt;1 : L3$ address[7:6] used as L3$ bank IDs.&lt;br&gt;<strong>sarbcf_csr_l3bankidhashdis</strong></td>
</tr>
<tr>
<td>29:28</td>
<td>RW</td>
<td>10b</td>
<td>Core</td>
<td><strong>Arbitration priority order between RCC and MSC (APORM):</strong>&lt;br&gt;Arbitration priority order between RCC and MSC.&lt;br&gt;00/11: Invalid; default setting used&lt;br&gt;10 : Default setting; RCC &lt; MSC (i.e., MSC has higher priority)&lt;br&gt;01: RCC &gt; MSC (i.e., RCC has higher priority)&lt;br&gt;<strong>sarbcf_csr_rcc_msc_pri[1:0]</strong></td>
</tr>
<tr>
<td>27:22</td>
<td>RW</td>
<td>100100b</td>
<td>Core</td>
<td><strong>Arbitration priority order between RCZ, STC, and HIZ (APORSH):</strong>&lt;br&gt;Arbitration priority order between RCZ, STC, and HIZ.&lt;br&gt;100100 : Default setting; RCZ &lt; STC &lt; HIZ&lt;br&gt;(i.e., RCZ has lowest priority; HIZ has highest priority)&lt;br&gt;100001 : RCZ &lt; HIZ &lt; STC&lt;br&gt;011000 : STC &lt; RCZ &lt; HIZ&lt;br&gt;010010 : STC &lt; HIZ &lt; RCZ&lt;br&gt;001001 : HIZ &lt; RCZ &lt; STC&lt;br&gt;000110 : HIZ &lt; STC &lt; RCZ&lt;br&gt;Note: Others settings are invalid, and result in use of default.&lt;br&gt;<strong>sarbcf_csr_rcz_stc_hiz_pri[5:0]</strong></td>
</tr>
<tr>
<td>21:19</td>
<td>RW</td>
<td>010b</td>
<td>Core</td>
<td><strong>Write data port arbitration priority between Z client writes and L3$ evictions (WDPAGAPZ):</strong>&lt;br&gt;Z Max Write Request Limit Count (GFXC_MRLC):&lt;br&gt;This is the MAX number of Allowed Requests Count - These counters keep track of the accepted requests from each engine. Requests are counted, regardless of kind of cycle (both Slice 0 and1). Minimum count value must be = 1&lt;br&gt;<strong>sarbcf_csr_wdpagapz[2:0]</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 18:16 | RW     | 010b          | Core    | **Write data port arbitration priority between C client writes and Z/L3$ writes/evictions (WDPAGAPC):**  
C Max Request Limit Count (GFXZ_MRLC):  
This is the MAX number of Allowed Requests Count - These counters keep track of the accepted requests from each engine. Requests are counted, regardless of kind of cycle (both Slice 0 and1). Minimum count value must be = 1  
sarbcf_csr_wdpagapc[2:0] |
| 15    |        |               |         |             |
| 14:12 | RW     | 100b          | Core    | **L3 Max Write Request Limit Count (GFXL3):**  
L3 Max Write Request Limit Count (GFXL3_MRLC):  
This is the MAX number of Allowed Requests Count - These counters keep track of the accepted requests from each engine. Requests are counted, regardless of kind of cycle (Miss/Hit/Present). Minimum count value must be = 1  
sarbcf_csr_wdpagapl3[2:0] |
| 11:9  | RW     | 010b          | Core    | **Vebox Max Request Limit Count (VEBOX):**  
Vebox Max Request Limit Count (VEBOX_MRLC):  
This is the MAX number of Allowed Requests Count - These counters keep track of the accepted requests from each engine. Requests are counted, regardless of kind of cycle (both Slice 0 and1). Minimum count value must be = 1  
sarbcf_csr_wdpagapv[2:0] |
| 8     | RW     | 1b            | Core    | **GAPs_fixarb_en (GAPSFXABEN):**  
Arbitration order adjustment  
sarbcf_csr_gaps_fixarb_en |
| 7:0   | RO     | 00h           | Core    | **Reserved (RSVD):** |

**L3SQCREG4 - L3 SQC register 4**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B034-B037h
**Default Value:** 08000000h  
**Access:** RWHC; RO; RW;  
**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>30</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>29:28</td>
<td>RO</td>
<td>00b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>27</td>
<td>RW</td>
<td>1b</td>
<td>Core</td>
<td><strong>L3SQ URB Read CAM Match Disable (SQRBRDCAMDIS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Disables the L3SQ Cam Match ability for URB Reads. By disabling, this allows</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a performance mode where URB reads are not dependent upon one another but</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>only on any previous URB writes to the same address. This allows many</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>URB reads to the same cacheline at any given time instead of serializing the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>requests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = URB Read CAM matching is disabled; multiple URB reads to the same</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cacheline are allowed to be concurrent(default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = URB Read CAM matching is enabled; multiple URB reads to the same</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cacheline are serialized</td>
</tr>
<tr>
<td>26</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td><strong>LSQC reset fcount (LSQCRFCNT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>self clearing register bit - Write to this register generates 1 clock pulse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbfc_csr_lsqc_rst_fcount to lsqc and also used to clear the register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbfc_csr_lsqc_rst_fcount_lvl is output of configdb.</td>
</tr>
<tr>
<td>25:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**SCRATCH1 - SCRATCH1**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B038-B03Bh  
**Default Value:** 00000000h  
**Access:** RW;  
**Size:** 32 bits
<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
| 30    | R/W    | 0b            | Core    | **RO Serialization Disable:**  
For 3D benchmarks, L3 introduced an optimization on B-step to enable back-to-back reads to the same RO space to streamline rather than waiting for each other to complete. This is a crucial performance fix.  
0: Enable optimization  
1: Disable optimization  
**NOTE:**  
*With RO serialization enabled, the following restrictions need to be followed:*  
- Priority selection should not be disabled (don’t set bit 16,15, 11,7,3) in L3SQCREG2 (0xb012). Keep the default value.  
- don’t disable aging (don’t set bit 19) in L3CNTLREG1 (0xb01c). Keep the default value. |
| 29    | RW     | 0b            | Core    | **RW Space Serialization Disable:**  
For 3D benchmarks, L3 introduced as an optimization on B-step to enable back-to-back reads to the same RW space to streamline rather than waiting for each other to complete. This is a crucial performance fix.  
0: Enable optimization  
1: Disable optimization |
| 28    | RW     | 0b            | Core    | Enable bank hanging for parity error.  
0: disable bank hanging  
1:enable bank hanging |
| 27    | RW     | 0b            | Core    | Disable data atomic in L3  
0: enable data atomic in L3  
1:disable data atomic in L3 |
<p>| 26:19 | RW     | 0000000000b   | Core    | SCRATCH (SCRATCH): |
| 18:4  | RW     | 0b            | Core    | Sub-Bank &amp; Row selection for error injection |</p>
<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non-SLM configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 18: unused</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bits 17:7: Row address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bits 6:4: Sub-Bank number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLM configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bits 18:7: Row address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bits 6:4: Sub-Bank number</td>
</tr>
<tr>
<td>3:2</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Bank ID for error injection</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00b: Inject in Bank 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01b: Inject in Bank 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10b: Inject in Bank 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11b: reserved – error injection not supported for SLM bank3.</td>
</tr>
<tr>
<td>1</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Slice ID for Error injection</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: Inject error in Slice 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: Inject error in Slice 1</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Parity Error Injection Enable</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: Disable parity error injection</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: Enable parity error injection</td>
</tr>
</tbody>
</table>

**LTIOREG - LTIO Register**

*B/D/F/Type:* 0/0/0/SARBunit_Config

*Address Offset:* B03C-B03Fh

*Default Value:* 00000086h

*Access:* RW; RO;

*Size:* 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>LTIO Arb Mode (LTOARBM):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0- default- optimized arbitration</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>30</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td>LTIO FCOUNT SEL (LTIOFSEL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0-default - pick hardware synchronized</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1- pick static fcount from sarb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_LTIO_arb_mode</td>
</tr>
<tr>
<td>29</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>28:24</td>
<td>RW</td>
<td>00000b</td>
<td>Core</td>
<td>FCOUNT (FCNT):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fcount</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_LTIO_fcount[4:0]</td>
</tr>
<tr>
<td>23:8</td>
<td>RO</td>
<td>0000h</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>7:4</td>
<td>RW</td>
<td>1000b</td>
<td>Core</td>
<td>LTIO WaterMark High (LTIOWMKH):</td>
</tr>
<tr>
<td>3:0</td>
<td>RW</td>
<td>0110b</td>
<td>Core</td>
<td>LTIO WaterMark Low (LTIOWMKL):</td>
</tr>
</tbody>
</table>

**CLMREDS0 - Column Redundancy Slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B044-B047h

**Default Value:** 00000000h

**Access:** RW; RO;

**Size:** 32 bits

This register is written by mbcunit and is not ctx saved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td>column redundancy bank 0 (CRB0): sarb_ltcd0_fuse[6:0]</td>
</tr>
<tr>
<td>24:18</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td>Column Redundancy bank1 (CRB1): sarb_ltcd1_fuse[6:0]</td>
</tr>
<tr>
<td>17:11</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td>column redundancy bank 2 (CRB2): sarb_ltcd2_fuse[6:0]</td>
</tr>
<tr>
<td>10:4</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td>column redundancy bank 3 (CRB3): sarb_ltcd3_fuse[6:0]</td>
</tr>
</tbody>
</table>
### LPCNTR1S0 - LPFC counter reg01 slice 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
</tbody>
</table>

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B04C-B04Fh

**Default Value:** 00000000h

**Access:** RO

**Size:** 32 bits

Counter 0

### LPCNTR2S0 - LPFC counter reg02 slice 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 1 slice 0 (LPFCCNT01): Counter 1</td>
</tr>
</tbody>
</table>

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B050-B053h

**Default Value:** 00000000h

**Access:** RO

**Size:** 32 bits

### LPCNTR3S0 - LPFC counter reg03 slice 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 2 slice 0 (LPFCCNT02): Counter2</td>
</tr>
</tbody>
</table>

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B054-B057h

**Default Value:** 00000000h

**Access:** RO

**Size:** 32 bits
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Core</td>
</tr>
</tbody>
</table>

**LPCNTR4S0 - LPFC counter reg04 slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B058-B05Bh
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Core</td>
</tr>
</tbody>
</table>

**LPCNTR5S0 - LPFC counter reg05 slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B05C-B05Fh
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Core</td>
</tr>
</tbody>
</table>

**LPCNTR6S0 - LPFC counter reg06 slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B060-B063h
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Core</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Counter6</td>
</tr>
</tbody>
</table>

**LPCNTR7S0 - LPFC counter reg07 slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B064-B067h
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 7 slice 0 (LPFCCNT07): Counter7</td>
</tr>
</tbody>
</table>

**LPCNTR8S0 - LPFC counter reg08 slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B068-B06Bh
- **Default Value:** 00000000h
- **Access:** RO;
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 8 slice 0 (LPFCCNT08): Counter8</td>
</tr>
</tbody>
</table>

**L3B0REG00 - L3 bank0 reg0 log error slice 0**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B070-B073h
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1   | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 0     | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

**L3B0REG01 - L3 bank0 reg1 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B074-B077h

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The number of rows vary between 4K vs 8K/16K subbanks which requires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The number of rows vary between 4K vs 8K/16K subbanks which requires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG02 - L3 bank0 reg2 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B078-B07Bh

**Default Value:** 00000000h
Access: WO; RO; 
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO | 000h | Core | **Row Number for Error1 (RNUMERR1):**
Row Number for Error1:
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
| 20:17 | RO | 0000b | Core | Reserved (RSVD): |
| 16 | WO | 0b | Core | **Valid Error 1 (VLDERR1):**
Valid Error:
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO | 000h | Core | **Row Number for Error0 (RNUMERR0):**
Row Number for Error0:
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
| 4:1 | RO | 0000b | Core | Reserved (RSVD): |
| 0 | WO | 0b | Core | **Valid Error 0 (VLDERR0):**
Valid Error:
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

**L3B0REG03 - L3 bank0 reg3 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Confi
Address Offset: B07C-B07Fh
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B0REG04 - L3 bank0 reg4 log error slice0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
|       |        |               |         | Row Number for Error1:  
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
|       |        |               |         | This field contains the row# with the error                                  |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):**                                                         |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
|       |        |               |         | Valid Error:  
|       |        |               |         | The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.                                      |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
|       |        |               |         | Row Number for Error0:  
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
|       |        |               |         | This field contains the row# with the error                                  |

**L3B0REG05 - L3 bank0 reg5 log error slice0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B084-B087h

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG06 - L3 bank0 reg6 log error slice 0**

**B/D/F/Type:**
0/0/0/SARBunit_Config

**Address Offset:**
B088-B08Bh

**Default Value:**
00000000h

**Access:**
WO; RO;

**Size:**
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error1 (RNUMERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 1 (VLDERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error0 (RNUMERR0):</td>
</tr>
</tbody>
</table>
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO | 000h | Core | **Row Number for Error1 (RNUMERR1):**  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO | 0000b | Core | **Reserved (RSVD):** |
| 16 | WO | 0b | Core | **Valid Error 1 (VLDERR1):** |

### L3B0REG07 - L3 Bank0 reg7 log error slice0

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B08C-B08Fh
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

### L3B1REG00 - L3 bank1 reg0 log error slice 0

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B090-B093h  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>
| 16   | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
|      |        |               |         | Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
|      |        |               |         | Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1  | RO     | 0000b         | Core    | **Reserved (RSVD):**  |
| 0    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
|      |        |               |         | Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

**L3B1REG01 - L3 bank1 reg1 log error slice 0**

**B/D/F/Type:** 0/0/0/SAR8unit_Confg  
**Address Offset:** B094-B097h  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits  

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.  
The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B1REG02 - L3 bank1 reg2 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B098-B09Bh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B1REG03 - L3 bank1 reg3 log error slice 0

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B09C-B09Fh
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong> Row Number for Error1: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong> Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong> Row Number for Error0: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong> Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B1REG05 - L3 bank1 reg5 log error slice 0

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B0A4-B0A7h

Default Value: 00000000h

Access: WO; RO;

Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**
|       |        |               |         | Row Number for Error1:
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
|       |        |               |         | This field contains the row# with the error                     |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):**                                            |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**
|       |        |               |         | Valid Error:
|       |        |               |         | The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**
|       |        |               |         | Row Number for Error0:
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
|       |        |               |         | This field contains the row# with the error                     |
| 4:1   | RO     | 0000b         | Core    | **Reserved (RSVD):**                                            |
| 0     | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**
<p>|       |        |               |         | Valid Error:                                                    |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong>&lt;br&gt;Row Number for Error1:&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong>&lt;br&gt;Valid Error:&lt;br&gt;The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong>&lt;br&gt;Row Number for Error0:&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong>&lt;br&gt;Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
**L3B1REG07 - L3 bank1 reg7 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0AC-B0AFh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B2REG00 - L3 bank2 reg0 log error slice 0

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B0B0-B0B3h

Default Value: 00000000h

Access: WO; RO;

Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
**L3B2REG02 - L3 bank2 reg2 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0B8-B0BBh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error1 (RNUMERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 1 (VLDERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error0 (RNUMERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
|       |        |               |         | Row Number for Error1:  
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
|       |        |               |         | This field contains the row# with the error                                   |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):**                                                       |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**                                               |
|       |        |               |         | Valid Error:  
|       |        |               |         | The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**                                     |
|       |        |               |         | Row Number for Error0:  
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
|       |        |               |         | This field contains the row# with the error                                   |
| 4:1   | RO     | 0000b         | Core    | **Reserved (RSVD):**                                                       |
| 0     | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**                                               |
|       |        |               |         | Valid Error:  

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B2REG05 - L3 bank2 reg5 log error slice 0

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B0C4-B0C7h

Default Value: 00000000h

Access: WO; RO;

Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO | 000h | Core | **Row Number for Error1 (RNUMERR1):**  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO | 0000b | Core | **Reserved (RSVD):** |
| 16 | WO | 0b | Core | **Valid Error 1 (VLDERR1):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO | 000h | Core | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1 | RO | 0000b | Core | **Reserved (RSVD):** |
| 0 | WO | 0b | Core | **Valid Error 0 (VLDERR0):**  
Valid Error: |
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B2REG06 - L3 bank2 reg6 log error slice 0

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0C8-B0CBh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong> Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. The field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong> Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong> Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. The field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong> Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has been detected. The number of rows vary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>between 4K vs 8K/16K subbanks which requires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>corresponding logical 16KB group should bypass</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has been detected. The number of rows vary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>between 4K vs 8K/16K subbanks which requires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B3REG02 - L3 bank3 reg2 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B0D8-B0DBh  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted. The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDErr1):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>
| 0   | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
  Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

### L3B3REG03 - L3 bank3 reg3 log error slice 0

**B/D/F/Type:**
0/0/0/SARBunit_Config

**Address Offset:**
B0DC-B0DFh

**Default Value:**
00000000h

**Access:**
WO; RO;

**Size:**
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
  Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 16   | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
  Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
<p>| 15:5 | WO     | 000h          | Core    | <strong>Row Number for Error0 (RNUMERR0):</strong> |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B3REG04 - L3 bank3 reg4 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0E0-B0E3h

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
</tbody>
</table>
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 15:5 | WO     | 000h          | Core    | Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
|      |        |               |         |             |
| 4:1  | RO     | 0000b         | Core    | Row Number for Error0 (RNUMERR0):  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
|      |        |               |         |             |
| 0    | WO     | 0b            | Core    | Valid Error 0 (VLDERR0):  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
|      |        |               |         |             |

### L3B3REG05 - L3 bank3 reg5 log error slice 0

**B/D/F/Type:**  
0/0/0/SARBunit_Config

**Address Offset:**  
B0E4-B0E7h

**Default Value:**  
00000000h

**Access:**  
WO; RO;

**Size:**  
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 31.21 | WO     | 000h          | Core    | Row Number for Error1 (RNUMERR1):  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. |
<p>| | | | | |
|      |        |               |         |             |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>

756
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):**                                                                                                                                 |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16 KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively.  
This field contains the row# with the error |

The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

**L3B3REG07 - L3 bank3 reg7 log error slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B0EC-B0EFh  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

Row Number for Error1 (RNUMERR1):  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively.  
This field contains the row# with the error

Reserved (RSVD):  

Valid Error 1 (VLDERR1):  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

Row Number for Error0 (RNUMERR0):  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively.  
This field contains the row# with the error
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**LPFCREG0 - First Buffer Size and Start**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0F0-B0F3h

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>RW</td>
<td>0000h</td>
<td>Core</td>
<td><strong>First Virtual Buffer Base (FVBB):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First Virtual Buffer Base: Programmed by driver to allocate a space for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>performance data storage. The buffer size should be aligned to the size of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the memory allocated so it naturally aligns to the base (i.e. for 128KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[16]=0, 256KB bit[17:16]=0, 512KB bit[18:16]=0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_virtual_base0[31:16]</td>
</tr>
<tr>
<td>15:12</td>
<td>RW</td>
<td>0000b</td>
<td>Core</td>
<td><strong>First Buffer Size (FBS):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First Buffer Size: Determines the allowed buffer size for performance data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000: 64KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001: 128KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010: 256KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011: 512KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111: 2GB</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>11:3</td>
<td>RO</td>
<td>000h</td>
<td>Core</td>
<td>Signal - sarbcf_lpfc_buffer_size0[3:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Frame Cnt and Draw Call Enable (FCDCE):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Frame Cnt and Draw Call Enable - If this mode is enabled then one of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>counters will be replaced by Frame Cnt and Draw call numbers to form the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>data packet to memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_csr_lpfc_framecnt_en</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>this bit is used by both slices</td>
</tr>
<tr>
<td>1</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Enable Dual-buffer mode (EDBM):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enable Dual-buffer mode: It enables the capability of h/w generating</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>interrupts to GFX driver to enable double buffering where memory content</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>can be dumped into hard-drive to enable usage of hard drive for performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_mode_sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>this bit is used by both slices</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>0b</td>
<td>Core</td>
<td><strong>Master Counter Enable (MCE):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Master Counter Enable: This is the global enable for performance tracking</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>once set, it will kick off all performance tracking mechanism.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_master_cnt_en</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>this bit is used by both slices</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Workaround: For LPFC to dump the contents of the last buffer (if buffer is not filled up), it needs to be disabled.</strong></td>
</tr>
</tbody>
</table>

**LPFCREG2 - Second Buffer Size**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B0F4-B0F7h
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>RW</td>
<td>0000h</td>
<td>Core</td>
<td><strong>Second Virtual Buffer Base (SVBB0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Second Virtual Buffer Base: Programmed by driver to allocate a space for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>performance data storage. The buffer size should be aligned to the size of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the memory allocated so it naturally aligns to the base (i.e. for 128KB bit[16]=0,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256KB bit[17:16]=0, 512KB bit[18:16]=0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_virtual_base1[31:16]</td>
</tr>
<tr>
<td>15:12</td>
<td>RW</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Second Buffer Size 0 (SBS0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Second Buffer Size: Determines the allowed buffer size for performance data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000: 64KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001: 128KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010: 256KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011: 512KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111: 2GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_buffer_size1[3:0]</td>
</tr>
<tr>
<td>11:0</td>
<td>RO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**LPFCREG03 - Error Reporting Reg Slice 0**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B0F8-B0FBh

**Default Value:** 00000000h

**Access:** RO; RWC;

**Size:** 32 bits

This register is not ctx saved/restored.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:5</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>4</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>First Content Buffer Ready 0 (FRSNTBFR0):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RST/PWR</td>
<td>First Content Buffer Ready: This bits gets set by the h/w when the buffer is completely filled up and cleared by the driver when the contents of this buffer is copied out of memory. Will be set by lpfc_sarbcf_buffer0_ready (pulse) sarbcf_lpfc_buffer0_ready (static signal to lpfc)</td>
</tr>
<tr>
<td>3</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>Second Content Buffer Ready slice 0 (SCNBFR0):</strong> Second Content Buffer Ready: This bits gets set by the h/w when the buffer is completely filled up and cleared by the driver when the contents of this buffer is copied out of memory. Will be set by lpfc_sarbcf_buffer1_ready(pulse) sarbcf_lpfc_buffer1_ready (static signal to lpfc)</td>
</tr>
<tr>
<td>2</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>Write Expired Error slice 0 (WEERR0):</strong> Write Expired Error: If DMA controller could not get a chance to push the write of 64Bytes to GAPL3 and data get clobbered with the new expiration of the save timer, this error bit will be set to indicate something went wrong. Signal - lpfc_sarbcf_wrexp_error</td>
</tr>
<tr>
<td>1</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>Buffer full Error Slice 0 (BFFLERR0):</strong> Set by lpfc_sarbcf_error_buffer_full When both buffers are full lpfc will set this bit or if only 1 buffer is enabled then lpfc will set this bit when the buffer is full.</td>
</tr>
<tr>
<td>0</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**LPFCREG04 - Frame count and Draw call number**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B0FC-B0FFh  
**Default Value:** 00000000h  
**Access:** RO; RW;  
**Size:** 32 bits
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>RO</td>
<td>0000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>15:8</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>Frame Number (FRMNUM):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Frame Number: This field is for GFX driver to populate the Frame number (or Frame color). It is an optional field for s/w to select the data samples that belongs to different frames. This 8-bit field can be incremented by the driver at every frame boundary. The frame number recording is an option with the counter selects. Signal - sarbcf_lpfc_frame_num[7:0]</td>
</tr>
<tr>
<td>7:0</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>Draw Call Number (DRWCLNUM):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Draw Call Number: It is theoretically possible for s/w to increment a count with each draw call submission, which in turn can be reported out as part of the data stream. Signal - sarbcf_lpfc_drawcall_num[7:0]</td>
</tr>
</tbody>
</table>

**LPFCREG05 - SAVE Timer**

<table>
<thead>
<tr>
<th>B/D/F/Type:</th>
<th>0/0/0/SARBunit_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Offset:</td>
<td>B100-B103h</td>
</tr>
<tr>
<td>Default Value:</td>
<td>00000000h</td>
</tr>
<tr>
<td>Access:</td>
<td>RO; RW;</td>
</tr>
<tr>
<td>Size:</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>30:29</td>
<td>RW</td>
<td>11b</td>
<td>Core</td>
<td><strong>Counter ENabling Selection (CNTRENSEL):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Counter Enabling Selection: Enables different number of counters aiding in data compression for focused studies 00 : Only Counter#0 is enabled 01 : Counters #0 &amp; #1 are enabled 10 : Counters #0, #1, #2 and #3 are enabled 11 : All counters are enabled</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>28:24</td>
<td>RO</td>
<td>00000b</td>
<td>Core</td>
<td>Signal - sarbcf_lpfc_cnt_enabled[1:0]</td>
</tr>
<tr>
<td>23:0</td>
<td>RW</td>
<td>001000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Save Timer Interval (SVTMRINT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Save Timer Interval: This is the interval for sampling the performance counters and writing to memory. Each time it expires, the counters will be sampled and packetized to be sent to DMA controller.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The minimum granularity of sampling period if 256clocks. The value in this register is used as 256 x value to find the sampling window. For a 1Ghz core clock it provides up to 4ns of sampling period while matching the maximum capability of the event counters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1-256clks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2-512clks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...so on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8-2048clks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - sarbcf_lpfc_savetimer_int[23:0]</td>
</tr>
</tbody>
</table>

**L3 Performance Counter Event Table**

<table>
<thead>
<tr>
<th>Client Encoding</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAFS Rd</td>
<td>00</td>
</tr>
<tr>
<td>GAFS Wr</td>
<td>01</td>
</tr>
<tr>
<td>HDC0 Data Rd</td>
<td>02</td>
</tr>
<tr>
<td>HDC0 Const Rd</td>
<td>03</td>
</tr>
<tr>
<td>HDC0 URB Rd</td>
<td>04</td>
</tr>
<tr>
<td>HDC0 Data Wr</td>
<td>05</td>
</tr>
<tr>
<td>HDC0 URB Wr</td>
<td>06</td>
</tr>
<tr>
<td>HDC1 Data Rd</td>
<td>07</td>
</tr>
<tr>
<td>HDC1 Const Rd</td>
<td>08</td>
</tr>
<tr>
<td>HDC1 URB Rd</td>
<td>09</td>
</tr>
<tr>
<td>HDC1 Data Wr</td>
<td>0A</td>
</tr>
<tr>
<td>HDC1 URB Wr</td>
<td>0B</td>
</tr>
<tr>
<td>TDL0 Rd</td>
<td>0C</td>
</tr>
<tr>
<td>TDL1 Rd</td>
<td>0D</td>
</tr>
<tr>
<td>Tex0 Rd</td>
<td>0E</td>
</tr>
<tr>
<td>Client Encoding</td>
<td>hex</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----</td>
</tr>
<tr>
<td>Tex1 Rd</td>
<td>0F</td>
</tr>
<tr>
<td>Tex2 Rd (reserved)</td>
<td>10</td>
</tr>
<tr>
<td>Tex3 Rd (reserved)</td>
<td>11</td>
</tr>
<tr>
<td>SBE Rd</td>
<td>12</td>
</tr>
<tr>
<td>IC0 Rd</td>
<td>13</td>
</tr>
<tr>
<td>IC1 Rd</td>
<td>14</td>
</tr>
<tr>
<td>SARB Rd</td>
<td>15</td>
</tr>
<tr>
<td>Aggregated Tex</td>
<td>16</td>
</tr>
<tr>
<td>SLM0 Rd</td>
<td>17</td>
</tr>
<tr>
<td>SLM1 Rd</td>
<td>18</td>
</tr>
<tr>
<td>SLM0 Wr</td>
<td>19</td>
</tr>
<tr>
<td>SLM1 Wr</td>
<td>1A</td>
</tr>
<tr>
<td>SLM0 Atomics</td>
<td>1B</td>
</tr>
<tr>
<td>SLM1 Atomics</td>
<td>1C</td>
</tr>
<tr>
<td>Reserved</td>
<td>1D</td>
</tr>
<tr>
<td>Reserved</td>
<td>1E</td>
</tr>
<tr>
<td>Reserved</td>
<td>1F</td>
</tr>
<tr>
<td>FF Stalls</td>
<td>20</td>
</tr>
<tr>
<td>HDC Stalls</td>
<td>21</td>
</tr>
<tr>
<td>TDL Stalls</td>
<td>22</td>
</tr>
<tr>
<td>Texture Stalls</td>
<td>23</td>
</tr>
<tr>
<td>IC Stalls</td>
<td>24</td>
</tr>
<tr>
<td>SBE Stalls</td>
<td>25</td>
</tr>
<tr>
<td>SLM Stalls</td>
<td>26</td>
</tr>
<tr>
<td>Bank0 Total Hits</td>
<td>40</td>
</tr>
<tr>
<td>Bank0 Total Cycles</td>
<td>41</td>
</tr>
<tr>
<td>Bank0 Total Rds</td>
<td>42</td>
</tr>
<tr>
<td>Bank0 Total Wrs</td>
<td>43</td>
</tr>
<tr>
<td>Bank0 FF Rds</td>
<td>44</td>
</tr>
<tr>
<td>Bank0 FF Wrs</td>
<td>45</td>
</tr>
<tr>
<td>Bank0 DC Rds</td>
<td>46</td>
</tr>
<tr>
<td>Bank0 DC Wrs</td>
<td>47</td>
</tr>
<tr>
<td>Bank0 DC Hits</td>
<td>48</td>
</tr>
<tr>
<td>rsvd</td>
<td>49</td>
</tr>
<tr>
<td>Bank0 Tex Rds</td>
<td>4A</td>
</tr>
<tr>
<td>Bank0 Tex Hits</td>
<td>AB</td>
</tr>
<tr>
<td>Client Encoding</td>
<td>hex</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>Bank0 IC Rds</td>
<td>AC</td>
</tr>
<tr>
<td>Bank0 IC Hits</td>
<td>4D</td>
</tr>
<tr>
<td>Reserved</td>
<td>4E</td>
</tr>
<tr>
<td>Reserved</td>
<td>4F</td>
</tr>
<tr>
<td>Bank1 Events</td>
<td>50-5F (except 59-reserved)</td>
</tr>
<tr>
<td>Bank2 Events</td>
<td>60-6F(except 69-reserved)</td>
</tr>
<tr>
<td>Bank3 Events</td>
<td>70-7F(except 79-reserved)</td>
</tr>
<tr>
<td>MSC Rd</td>
<td>80</td>
</tr>
<tr>
<td>MSC Wr</td>
<td>81</td>
</tr>
<tr>
<td>STC Rd</td>
<td>82</td>
</tr>
<tr>
<td>STC Wr</td>
<td>83</td>
</tr>
<tr>
<td>Hiz Rd</td>
<td>84</td>
</tr>
<tr>
<td>Hiz Wr</td>
<td>85</td>
</tr>
<tr>
<td>RCZ Rd</td>
<td>86</td>
</tr>
<tr>
<td>RCZ Wr</td>
<td>87</td>
</tr>
<tr>
<td>RCC Rd</td>
<td>88</td>
</tr>
<tr>
<td>RCC Wr</td>
<td>89</td>
</tr>
<tr>
<td>Frame Number</td>
<td>F0</td>
</tr>
<tr>
<td>Draw call number</td>
<td>F1</td>
</tr>
</tbody>
</table>

**LPFCREG06 - Event selection and base counters**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B104-B107h  
**Default Value:** 00000000h  
**Access:** RW;  
**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:24 | RW     | 00h           | Core    | **Counter 7 client (CNT7CL):**  
sarbcf_lpfc_cnt7_client[7:0]  
Counter#0 Client Selection: This field controls which client’s request stream will be observed in counter#0 |
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Please refer to the L3 Performance Counter Event Table for definition of encodings.</td>
</tr>
</tbody>
</table>
| 23:16 | RW     | 00h           | Core    | **Counter 6 client (CNT6CL):**  
sarbcf_LPfc_cnt6_client[7:0]  
Counter#1 Client Selection: This field controls which client's request stream will be observed in counter#1  
Please refer to the L3 Performance Counter Event Table for definition of encodings. |
| 15:8  | RW     | 00h           | Core    | **Counter 5 client (CNT5CL):**  
sarbcf_LPfc_cnt5_client[7:0]  
Counter#2 Client Selection: This field controls which client's request stream will be observed in counter#2  
Please refer to the L3 Performance Counter Event Table for definition of encodings. |
| 7:0   | RW     | 00h           | Core    | **Counter 4 client (CNT4CL):**  
sarbcf_LPfc_cnt4_client[7:0]  
Counter#3 Client Selection: This field controls which client's request stream will be observed in counter#3  
Please refer to the L3 Performance Counter Event Table for definition of encodings. |
**LPFCREG07 - Event Selection and Base Counters**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B108-B10Bh
- **Default Value:** 00000000h
- **Access:** RW
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:24 | RW     | 00h           | Core    | **Counter 3 Client (CNT3CL):**  
|       |        |               |         | sarbcf_lpfc_cnt3_client[7:0]  
|       |        |               |         | Counter#4 Client Selection: This field controls which client's request stream will be observed in counter#4  
|       |        |               |         | Please refer to the *L3 Performance Counter Event Table* for definition of encodings. |
| 23:16 | RW     | 00h           | Core    | **Counter 2 Client (CNT2CL):**  
|       |        |               |         | sarbcf_lpfc_cnt2_client[7:0]  
|       |        |               |         | Counter#5 Client Selection: This field controls which client's request stream will be observed in counter#5  
|       |        |               |         | Please refer to the *L3 Performance Counter Event Table* for definition of encodings. |
| 15:8  | RW     | 00h           | Core    | **Counter 1 Client (CNT1CL):**  
|       |        |               |         | sarbcf_lpfc_cnt1_client[7:0]  
|       |        |               |         | Counter#6 Client Selection: This field controls which client's request stream will be observed in counter#6  
|       |        |               |         | Please refer to the *L3 Performance Counter Event Table* for definition of encodings. |
| 7:0   | RW     | 00h           | Core    | **Counter 0 Client (CNT0CL):**  
|       |        |               |         | sarbcf_lpfc_cnt0_client[7:0]  
|       |        |               |         | Counter#7 Client Selection: This field controls which client's request stream will be observed in counter#7  
|       |        |               |         | Please refer to the *L3 Performance Counter Event Table* for definition of encodings. |
### LPFCREG08 - MASTER start Timer

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RW</td>
<td>00000001h</td>
<td>Core</td>
<td>Master Start Timer (MSTSTTMR): sarbcf_lpfc_master_start_timer[31:0] So many clocks are expired before starting the rest of the counters. Time to wait is 256 * value clocks value for this register cannot be 0.</td>
</tr>
</tbody>
</table>

### L3SYNC - L3 Cross Sync Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td>Cross SYNC of L3 (CRSYNL3): Cross SYNC of L3: This is a message bit written by the cross L3 in case of GT4. To set this bit both bit[0] and bit[16] (mask) needs to be set. This bit stays set until the targeted L3 sends its SYNC FLUSH COMPLETE EVENT to HDC.</td>
</tr>
</tbody>
</table>
### slmmg - slm context save/restore msg

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B1F8-B1FBh

**Default Value:** 00000000h

**Access:** RO; RWHC;

**Size:** 32 bits

This register is written by tsg unit and is not ctx saved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:10</td>
<td>RO</td>
<td>000000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong> Reserved</td>
</tr>
<tr>
<td>9:2</td>
<td>RO</td>
<td>00h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>1</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td><strong>SLM restore msg (SLMRSTR):</strong> Restore - When set, TSG is requesting for restore of SLM from the address provided.</td>
</tr>
<tr>
<td>0</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td><strong>slm save msg (SLMSVMSG):</strong> SAve - When set, TSG is requesting for save of SLM to the address provided.</td>
</tr>
</tbody>
</table>

### SARBCSR - SARB config save msg

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B1FC-B1FFh

**Default Value:** 00000000h

**Access:** RWHC; RO;

**Size:** 32 bits

This register is not context saved and is written by cs unit

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:2</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>1</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td><strong>Context restore ack (CTXRSTRACK):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A write from cs to this bit along with mask bit 17 will prompt srb to ack ctx restore ack .</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>srb_ctx_restore - ctx restore from cs</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>RWHC</td>
<td>0b</td>
<td>Core</td>
<td>clr_sarb_ctx_restore - sARB clr this bit.</td>
</tr>
</tbody>
</table>

**Context save bit (SARBCS):**

A write from cs to this bit along with mask bit 16 will prompt sARB to start context save to cs.

sarb_ctx_save - ctx save from cs
clr_sarb_ctx_save - sARB clr this bit once ctx save sm kicks in.

---

**SARERRST1 - SARB Error Status slice1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B204-B207h

**Default Value:** 00000000h

**Access:** RO;

**Size:** 32 bits

Reports the error if any has occured for certain sARB features.

This register is not ctx saved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31  | RO     | 0b            | Core    | **Error if general bound is zero (ERRGENBDZO):**
Error if general bound is zero set by sARBunit
1: general bound address is 0
sarbcf_csr_gen_bnd_zero_err |
| 30  | RO     | 0b            | Core    | **Error if dynamic bound is zero (ERRDYNBDZO):**
Error if dynamic bound is zero- set by sARBunit
0: no error
1: dynamic address is 0
sarbcf_csr_dyn_bnd_zero_err |
<p>| 29  | RO     | 0b            | Core    | <strong>Reserved (RSVD):</strong> |
| 28  | RO     | 0b            | Core    | <strong>General Bound Check Overflow Error (GENBNDOVF):</strong> |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>General Bound Check Overflow Error - set by sarbunit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: overflow for general bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_gen_bnd_ovflw_err</td>
</tr>
<tr>
<td>27</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Dynamic Bound Check Overflow Error (DYNBDOVF):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dynamic Bound Check Overflow Error - set by sarbunit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: overflow for dynamic bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_dyn_bnd_ovflw_err</td>
</tr>
<tr>
<td>26</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Lower Bound Check Overflow Error (LWRBDOVF):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lower Bound Check Overflow Error - set by sarbunit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lower bound overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_lower_bnd_err</td>
</tr>
<tr>
<td>25:21</td>
<td>RO</td>
<td>00000b</td>
<td>Core</td>
<td><strong>INVALIDATION FLUSH STATUS REPORTING (INVSTRPT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>invalidation status for l3 is reported in this register.</td>
</tr>
<tr>
<td>20:18</td>
<td>RO</td>
<td>000b</td>
<td>Core</td>
<td><strong>SARB invalidation Status reporting (SARBINVSTRPT):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>invalidation status of sarb is reported in this register.</td>
</tr>
<tr>
<td>17</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>HW surface Bound Check Overflow Error (HWSBDOVF):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_hw_surf_bnd_ovflw_err</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HW Surface Bound Check Overflow Error - set by sarbunit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: overflow for bound check</td>
</tr>
<tr>
<td>16</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Error if hw surface bound is zero (ERRHWSNZO):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_csr_hw_surf_bnd_zero_err</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Error if hw surface bound is zero - set by sarbunit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: no error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: address is 0</td>
</tr>
<tr>
<td>15</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>buffer Ready intp err (INTPERR):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When both buffers are ready before one buffer ready is cleared by sft sarb will generate intp err (it is not expected that second buffer ready should assert while first buffer ready was not cleared by sftwr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_both_buffer_rd_intp_err</td>
</tr>
<tr>
<td>14:0</td>
<td>RO</td>
<td>0000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>----------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>31:25</td>
<td>RO</td>
<td>00000000b</td>
<td>Core</td>
<td>Reserved (RSVD): Reserved</td>
</tr>
<tr>
<td>24:14</td>
<td>RWC</td>
<td>00000000000b</td>
<td>Core</td>
<td>Parity row address error (PRTYROWNUM): Data array address which has parity B1: Report the data array address which has the Error \ltcd_sarb_parity_err_rownun[10:0] Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>13</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>Parity Error Valid (PRTYERRVLD): Parity Error valid Report the Parity Error \ltcd_sarb_parity_err_valid Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>12:11</td>
<td>RWC</td>
<td>00b</td>
<td>Core</td>
<td>Parity error bank number (PRTYBNKNUM): bank number which has parity error Report the bank no. which has the Error \ltcd_sarb_parity_err_banknum[1:0] Once set by HW, it can be cleared only by MMIO Write of 1 to this register bit 13.</td>
</tr>
<tr>
<td>10:8</td>
<td>RWC</td>
<td>000b</td>
<td>Core</td>
<td>Parity Error sub-bank no (PRTYSBNKNUM):</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>7</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td>RESERVED (RSVD): reserved</td>
</tr>
<tr>
<td>6:0</td>
<td>RO</td>
<td>00h</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
</tbody>
</table>

### L3CDERRST12 - L3CD Error Status register 2 slice 1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B20C-B20Fh

**Default Value:** 00000000h

**Access:** RO; RWC;

**Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:29</td>
<td>RO</td>
<td>000b</td>
<td>Core</td>
<td>Reserved (RSVD): reserved</td>
</tr>
</tbody>
</table>
| 28   | RWC   | 0b           | Core    | **URB High Limit Error on B3 (URBHLB3):**  
URB High Limit Error on B3:  
Report the URB High Limit Error- Address Bound check  
Once set, it can be cleared only by MMIO Write to this register. A write of value 1 will clear it  
( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in the MMIO as Error status. This can be only cleared by MMIO Write to that Bit. )  
litt3_sarb_urboff_error |
| 27   | RWC   | 0b           | Core    | **URB High Limit Error on B2 (URBHLB2):**  
URB High Limit Error on B2:  
Report the URB High Limit Error- Address Bound check  
Once set, it can be cleared only by MMIO Write to this register.  
( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in |
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RST/PWR</td>
<td>the MMIO as Error status. This can be only cleared by MMIO Write to that Bit.</td>
</tr>
<tr>
<td>26</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>ltcc2_sarb_urboff_error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RST/PWR</td>
<td><strong>URB High Limit Error on B1 (URBHLB1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>URB High Limit Error on B1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>Report the URB High Limit Error - Address Bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>Once set, it can be cleared only by MMIO Write to this register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in the MMIO as Error status. This can be only cleared by MMIO Write to that Bit. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>ltcc1_sarb_urboff_error</td>
</tr>
<tr>
<td>25</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>ltcc1_sarb_urboff_error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td><strong>URB High Limit Error on B0 (URBHLB0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>URB High Limit Error on B0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>Report the URB High Limit Error - Address Bound check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>Once set, it can be cleared only by MMIO Write to this register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>( LTCC generates a Pulse to SARB Config, Sarb Config sets and reflect it in the MMIO as Error status. This can be only cleared by MMIO Write to that Bit. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
<td>ltcc0_sarb_urboff_error</td>
</tr>
<tr>
<td>24:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**CLMREDS1 - Column Redundancy Slice 1**

B/D/F/Type: 0/0/0/SARBunit_Config

Address Offset: B244-B247h

Default Value: 00000000h

Access: RW; RO;

Size: 32 bits

This register is written by mbcunit and is not ctx saved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>column redundancy bank 0 (CRB0):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>sarb_ltcd0_fuse[6:0]</strong></td>
</tr>
<tr>
<td>24:18</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>column redundancy bank 1 (CRB1):</strong> sarb_ltcd1_fuse[6:0]</td>
</tr>
<tr>
<td>17:11</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>column redundancy bank 2 (CRB2):</strong> sarb_ltcd2_fuse[6:0]</td>
</tr>
<tr>
<td>10:4</td>
<td>RW</td>
<td>00h</td>
<td>Core</td>
<td><strong>column redundancy bank 3 (CRB3):</strong> sarb_ltcd3_fuse[6:0]</td>
</tr>
<tr>
<td>3:0</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**LPCNTR1S1 - LPFC counter reg01 slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B24C-B24Fh
- **Default Value:** 00000000h
- **Access:** RO
- **Size:** 32 bits

Counter 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>LPFC Counter 1 slice 1 (LPFCCNT11):</strong> Counter 1</td>
</tr>
</tbody>
</table>

**LPCNTR2S1 - LPFC counter reg02 slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B250-B253h
- **Default Value:** 00000000h
- **Access:** RO
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td><strong>LPFC Counter 2 slice 1 (LPFCCNT12):</strong> counter2</td>
</tr>
</tbody>
</table>

775
LPCNTR3S1 - LPFC counter reg03 slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B254-B257h
Default Value: 00000000h
Access: RO;
Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 3 slice 1 (LPFCCNT13): counter3</td>
</tr>
</tbody>
</table>

LPCNTR4S1 - LPFC counter reg04 slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B258-B25Bh
Default Value: 00000000h
Access: RO;
Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 4 slice 1 (LPFCCNT14): counter4</td>
</tr>
</tbody>
</table>

LPCNTR5S1 - LPFC counter reg05 slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B25C-B25Fh
Default Value: 00000000h
Access: RO;
Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 5 slice 1 (LPFCCNT15): Counter5</td>
</tr>
</tbody>
</table>

LPCNTR6S1 - LPFC counter reg06 slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B260-B263h
Default Value: 00000000h
Access: RO;
Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 6 slice 1 (LPFCCNT16): Counter6</td>
</tr>
</tbody>
</table>

**LPCNTR7S1 - LPFC counter reg07 slice 1**

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B264-B267h
Default Value: 00000000h
Access: RO;
Size: 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>LPFC Counter 7 slice 1 (LPFCCNT17): counter7</td>
</tr>
</tbody>
</table>

**L3B0REG10 - L3 bank0 reg0 log error slice 1**

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B270-B273h
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO | 000h | Core | **Row Number for Error1 (RNUMERR1):**
Row Number for Error1:
The physical row number where the parity error has been detected. The
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error 0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10 bits vs 11 bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG11 - L3 bank0 reg1 log error slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B274-B277h
- **Default Value:** 0000000h
- **Access:** WO; RO
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong>&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong>&lt;br&gt;The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong>&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong>&lt;br&gt;The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG12 - L3 bank0 reg2 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B278–B27Bh  
**Default Value:** 00000000h  
**Access:** RO; WO;  
**Size:** 32 bits
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG13 - L3 bank0 reg3 log error slice 1**

<table>
<thead>
<tr>
<th>B/D/F/Type:</th>
<th>0/0/0/SARBunit_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Offset:</td>
<td>B27C-B27Fh</td>
</tr>
<tr>
<td>Default Value:</td>
<td>00000000h</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B0REG14 - L3 bank0 reg4 log error slice1**

**B/D/F/Type:** 0/0/0/SARBunit_Confi
**Address Offset:** B280-B283h  
**Default Value:** 00000000h  
**Access:** RO; WO;  
**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B0REG15 - L3 bank0 reg5 log error slice1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B284-B287h
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

### L3B0REG16 - L3 bank0 reg6 log error slice 1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21| WO     | 000h          | Core    | Row Number for Error1 (RNUMERR1):
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
| 20:17| RO     | 0000b         | Core    | Reserved (RSVD): |
| 16   | WO     | 0b            | Core    | Valid Error 1 (VLDERR1):
Valid Error:
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | Row Number for Error0 (RNUMERR0):
Row Number for Error0:
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
</tbody>
</table>

### L3B0REG17 - L3 bank0 reg7 log error slice1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B28C-B28Fh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
</tbody>
</table>
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

#### L3B1REG10 - L3 bank1 reg0 log error slice 1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B290-B293h

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|     |        |               |         | **Valid Error:**  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1  | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 0    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
### L3B1REG12 - L3 bank1 reg2 log error slice 1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B298-B29Bh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 16   | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1  | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 0    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
Valid Error: |
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
**L3B1REG13 - L3 bank1 reg3 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B29C-B29Fh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Row Number for Error1:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
</tbody>
</table>

|       |        |               |         | **Valid Error 1 (VLDERR1):**                           |
|       |        |               |         | The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):**                                  |

| 16    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**                          |
|       |        |               |         | The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**                 |
|       |        |               |         | This field contains the row# with the error            |
|       |        |               |         | **Row Number for Error0:**                            |
|       |        |               |         | The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. |

| 4:1   | RO     | 0000b         | Core    | **Reserved (RSVD):**                                  |

| 0     | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**                          |
|       |        |               |         | Valid Error:                                          |
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

## L3B1REG14 - L3 bank1 reg4 log error slice 1

### B/D/F/Type:
0/0/0/SARBunit_Config

### Address Offset:
B2A0-B2A3h

### Default Value:
00000000h

### Access:
WO; RO;

### Size:
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong>&lt;br&gt;Row Number for Error1:&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong>&lt;br&gt;Valid Error:&lt;br&gt;The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong>&lt;br&gt;Row Number for Error0:&lt;br&gt;The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.&lt;br&gt;This field contains the row# with the error</td>
</tr>
</tbody>
</table>
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
</tbody>
</table>
| 0   | WO     | 0b            | Core    | Valid Error 0 (VLDERR0):  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

#### L3B1REG15 - L3 bank1 reg5 log error slice 1

**B/D/F/Type:**
0/0/0/SARBunit_Config

**Address Offset:**
B2A4-B2A7h

**Default Value:**
00000000h

**Access:**
WO; RO;

**Size:**
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21| WO     | 000h          | Core    | Row Number for Error1 (RNUMERR1):  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17| RO     | 0000b         | Core    | Reserved (RSVD): |
| 16   | WO     | 0b            | Core    | Valid Error 1 (VLDERR1):  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
<p>| 15:5 | WO     | 000h          | Core    | Row Number for Error0 (RNUMERR0): |</p>
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B1REG16 - L3 bank1 reg6 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B2A8-B2ABh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error1 (RNUMERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 1 (VLDERR1):</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B1REG17 - L3 bank1 reg7 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B2AC-B2AFh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>
| 16   | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1  | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 0    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

L3B2REG11 - L3 bank2 reg1 log error slice 1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>\textbf{Row Number for Error1 (RNUMERR1):} \textbf{Row Number for Error1}: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>\textbf{Reserved (RSVD):}</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>\textbf{Row Number for Error0 (RNUMERR0):} \textbf{Row Number for Error0}: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
</tbody>
</table>

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B2REG12 - L3 bank2 reg2 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B2B8-B2BBh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
</tbody>
</table>
### Bit Access Default Value Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Row Number for Error0:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

### L3B2REG13 - L3 bank2 reg3 log error slice 1

#### B/D/F/Type:
0/0/0/SARBunit_Config

#### Address Offset:
B2BC-B2BFh

#### Default Value:
00000000h

#### Access:
WO; RO;

#### Size:
32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B2REG14 - L3 bank2 reg4 log error slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B2C0-B2C3h
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0: The physical row number where the parity error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has been detected. The number of rows vary between 4K vs 8K/16K subbanks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
L3B2REG15 - L3 bank2 reg5 log error slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B2C4-B2C7h
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted. The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B2REG16 - L3 bank2 reg6 log error slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B2C8-B2CBh
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:21 | WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**
Row Number for Error1:
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 16    | WO     | 0b            | Core    | **Valid Error 1 (VLDERR1):**
Valid Error:
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5  | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**
Row Number for Error0:
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.
This field contains the row# with the error |
### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 0 (VLDERR0):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

#### L3B2REG17 - L3 bank2 reg7 log error slice 1

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B2CC-B2CFh  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits  

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.  

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

### Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error1 (RNUMERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td>Valid Error 1 (VLDERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td>Row Number for Error0 (RNUMERR0):</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 31:21 | WO     | 000h          | Core    | Row Number for Error1 (RNUMERR1):  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 20:17 | RO     | 0000b         | Core    | Reserved (RSVD):  |
| 16    | WO     | 0b            | Core    | Valid Error 1 (VLDERR1):  |

L3B3REG10 - L3 bank3 reg0 log error slice 1

B/D/F/Type: 0/0/0/SARBunit_Config
Address Offset: B2D0-B2D3h
Default Value: 00000000h
Access: WO; RO;
Size: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
|      |        |               |         | **Valid Error:**  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |
| 15:5 | WO     | 000h          | Core    | **Row Number for Error0 (RNUMERR0):**  
Row Number for Error0:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.  
This field contains the row# with the error |
| 4:1  | RO     | 0000b         | Core    | **Reserved (RSVD):** |
| 0    | WO     | 0b            | Core    | **Valid Error 0 (VLDERR0):**  
Valid Error:  
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row. |

**L3B3REG11 - L3 bank3 reg1 log error slice 1**

**B/D/F/Type:** 0/0/0/SARBunit_Config  
**Address Offset:** B2D4-B2D7h  
**Default Value:** 00000000h  
**Access:** WO; RO;  
**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31.21| WO     | 000h          | Core    | **Row Number for Error1 (RNUMERR1):**  
Row Number for Error1:  
The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. |
<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>

Valid Error:

Validation
### L3B3REG13 - L3 bank3 reg3 log error slice 1

**B/D/F/Type:** 0/0/0/SARBunit_Config

**Address Offset:** B2DC-B2DFh

**Default Value:** 00000000h

**Access:** WO; RO;

**Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong> Row Number for Error1: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong> Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong> Row Number for Error0: The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error</td>
</tr>
</tbody>
</table>
### Bit 4:1
- **Access**: RO
- **Default Value**: 0000b
- **RST/PWR**: Core
- **Description**: Reserved (RSVD): The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

### Bit 0
- **Access**: WO
- **Default Value**: 0b
- **RST/PWR**: Core
- **Description**: Valid Error 0 (VLDERR0): The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

---

### L3B3REG14 - L3 bank3 reg4 log error slice 1

**B/D/F/Type**: 0/0/0/SARBunit_Config

**Address Offset**: B2E0-B2E3h

**Default Value**: 00000000h

**Access**: WO; RO;

**Size**: 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted. The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

---

### Bit 31:21
- **Access**: WO
- **Default Value**: 000h
- **RST/PWR**: Core
- **Description**: Row Number for Error1 (RNUMERR1): The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively. This field contains the row# with the error.

### Bit 20:17
- **Access**: RO
- **Default Value**: 0000b
- **RST/PWR**: Core
- **Description**: Reserved (RSVD): The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

### Bit 16
- **Access**: WO
- **Default Value**: 0b
- **RST/PWR**: Core
- **Description**: Valid Error 1 (VLDERR1): The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

### Bit 15:5
- **Access**: WO
- **Default Value**: 000h
- **RST/PWR**: Core
- **Description**: Row Number for Error0 (RNUMERR0):
## Bit Access Default Value RST/PWR Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error: The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B3REG15 - L3 bank3 reg5 log error slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B2E4-B2E7h
- **Default Value:** 00000000h
- **Access:** WO; RO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>

**L3B3REG16 - L3 bank3 reg6 log error slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B2E8-B2EBh
- **Default Value:** 00000000h
- **Access:** RO; WO;
- **Size:** 32 bits

The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The number of rows vary between 4K vs 8K/16K subbanks which requires 10bits vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.</td>
</tr>
</tbody>
</table>
The ERROR LOG registers of L3 will maintain the bad row information for each of the 16KB subbank groups. The LOG will be programmed by driver before any workloads are submitted.

The contents of the LOG register will be context Save&Restored by h/w around rc6 events.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error1 (RNUMERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>20:17</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>16</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 1 (VLDERR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The error located in field 15:5 is valid and corresponding logical 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>group should bypass this row.</td>
</tr>
<tr>
<td>15:5</td>
<td>WO</td>
<td>000h</td>
<td>Core</td>
<td><strong>Row Number for Error0 (RNUMERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Row Number for Error0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The physical row number where the parity error has been detected. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of rows vary between 4K vs 8K/16K subbanks which requires 10bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vs 11bits respectively.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This field contains the row# with the error</td>
</tr>
<tr>
<td>4:1</td>
<td>RO</td>
<td>0000b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Valid Error 0 (VLDERR0):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid Error:</td>
</tr>
</tbody>
</table>
The error located in field 15:5 is valid and corresponding logical 16KB group should bypass this row.

**LPFCREG13 - Error Reporting Reg Slice 1**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:5</td>
<td>RO</td>
<td>00000000h</td>
<td>Core</td>
<td>Reserved (RSVD):</td>
</tr>
<tr>
<td>4</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>First Content Buffer Ready 1 (FRSNTBFR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First Content Buffer Ready: This bits gets set by the h/w when the buffer is completely filled up and cleared by the driver when the contents of this buffer is copied out of memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Will be set by lpfc_sarbcf_buffer0_ready (pulse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_lpfc_buffer0_ready (static signal to lpfc)</td>
</tr>
<tr>
<td>3</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>Second Content Buffer Ready slice 1 (SCNBFR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Second Content Buffer Ready: This bits gets set by the h/w when the buffer is completely filled up and cleared by the driver when the contents of this buffer is copied out of memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Will be set by lpfc_sarbcf_buffer1_ready(pulse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sarbcf_lpfc_buffer1_ready (static signal to lpfc)</td>
</tr>
<tr>
<td>2</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td>Write Expired Error slice 1 (WEERR1):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write Expired Error: If DMA controller could not get a chance to push the write of 64Bytes to GAPL3 and data get clobbered with the new expiration of the save timer, this error bit will be set to indicate something went wrong.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Signal - lpfc_sarbcf_wrexp_error</td>
</tr>
<tr>
<td>Bit</td>
<td>Access</td>
<td>Default Value</td>
<td>RST/PWR</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>RWC</td>
<td>0b</td>
<td>Core</td>
<td><strong>Buffer full Error Slice 1 (BFFLERR1):</strong>&lt;br&gt;Set by lpfc_sarbcf_error_buffer_full&lt;br&gt;When both buffers are full lpfc will set this bit or if only 1 buffer is enabled then lpfc will set this bit when the buffer is full.</td>
</tr>
<tr>
<td>0</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Reserved (RSVD):</strong></td>
</tr>
</tbody>
</table>

**LPCNTR8S1 - LPFC counter reg08 slice 1**

- **B/D/F/Type:** 0/0/0/SARBunit_Config
- **Address Offset:** B268-B26Bh
- **Default Value:** 00000000h
- **Access:** RO
- **Size:** 32 bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/PWR</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:0| RO     | 00000000h     | Core    | **LPFC Counter 8 slice 1 (LPFCCNT18):**
Counter8 |
Media GPGPU Pipeline

GPGPU Overview

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 N0.

GPGPU Thread Limits

GPGPU requires 1024 SIMD channels to be available for a maximum size thread group. In a HSW/GT2 system with 10 EUs per subslice, each with 7 hardware threads, this means that a maximum size thread group will fit in a subslice if SIMD16 instructions are used, but not if SIMD8 are used. These limits can be circumvented for thread groups which do not need access to a barrier or SLM, in which case the thread group can cross sub-slices.

The Configurations section should be referenced to determine what SIMD is required to fit in the subslice of the targeted configuration.
GPGPU Commands

This section contains various commands for GPGPU, including a number of them shared with media mode.

MEDIA_VFE_STATE with varying definitions for different generations and projects:

**MEDIA_VFE_STATE**

**MEDIA_CURBE_LOAD**

**MEDIA_INTERFACE_DESCRIPTOR_LOAD**

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address, with varying definitions for different generations and projects:

**INTERFACE_DESCRIPTOR_DATA**

<table>
<thead>
<tr>
<th>Project:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

**MEDIA_STATE_FLUSH**

**GPGPU_WALKER**

**GPGPU_OBJECT**
**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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU_OBJECT/WALKER</td>
<td>Either a set of objects or a walker to dispatch a thread group which will write the next group’s properties to memory.</td>
</tr>
<tr>
<td>MI_FLUSH</td>
<td>Make sure the thread group has finished executing.</td>
</tr>
<tr>
<td>MEDIA_CURBE_LOAD</td>
<td>Load the thread ids for new group.</td>
</tr>
<tr>
<td>MI_LOAD_REGISTER_MEMORY</td>
<td>Load the indirect MMIO GPGPU registers from the mem written by the previous group.</td>
</tr>
<tr>
<td>GPGPU_WALKER (indirect)</td>
<td>A walker with the indirect bit set.</td>
</tr>
</tbody>
</table>

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 | Project |
      | 4     |         |
   b. GPGPU_DISPATCHDIMY - the Y dimension of the number of thread groups to dispatch in:
      | DWord | Project |
      | 6     |         |
   c. GPGPU_DISPATCHDIMZ - the Z dimension of the number of thread groups to dispatch in:
      | DWord | Project |
      | 8     |         |

See vol1c Memory Interface and Command Stream for the MMIO register addresses and formats.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Security:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WA.Workaround</td>
</tr>
</tbody>
</table>

The indirect registers are not supposed to be set to 0, but sometimes the kernel computing the value wants no work done and sets them to 0. This does not work correctly, so a workaround in the command stream is needed:

GPGPU_WALKER // The thread group which writes the indirect values to memory locations
MI_CONDITIONAL_BATCH_BUFFER_END DIMX StartX // End batch buffer if X dim in memory
  = StartX in DWord per table immediately below
MI_CONDITIONAL_BATCH_BUFFER_END DIMY 0 StartY // End batch buffer if Y dim in memory
  = StartY in DWord per table immediately below
MI_CONDITIONAL_BATCH_BUFFER_END DIMZ 0 StartZ // End batch buffer if Z dim in memory

821
<table>
<thead>
<tr>
<th>Project:</th>
<th>Security:</th>
</tr>
</thead>
<tbody>
<tr>
<td>WA.Workaround</td>
<td></td>
</tr>
</tbody>
</table>

= StartZ in DWord per table immediately below

```
MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMX DIMX // Normal load of register from memory
MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMY DIMY
MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMZ DIMZ
GPGPU_WALKER                                  // The thread groups which depend on the indirect dimensions
```

<table>
<thead>
<tr>
<th>Project</th>
<th>DWord Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>StartX in DW3, StartY in DW5, StartZ in DW7</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Command Ring</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI_SET_CONTEXT</td>
<td>Go to GPGPU context</td>
</tr>
<tr>
<td>MI_BATCHBUFFER_START</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command Batch</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU_OBJECT</td>
<td></td>
</tr>
<tr>
<td>GPGPU_OBJECT</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>(more threads forming a complete thread group)</td>
</tr>
<tr>
<td>MEDIA_STATE_FLUSH</td>
<td>Check for preemption at thread group boundary. &quot;Preemption&quot; defined by the UHPTR valid bit set.</td>
</tr>
<tr>
<td>MI_ARB_CHECK</td>
<td>Move the head only if UHPTR valid bit is set.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>GPGPU_WALKER</td>
<td></td>
</tr>
<tr>
<td>MEDIA_STATE_FLUSH</td>
<td>Check for preemption at thread group boundary internal to GPGPU_WALKER command. &quot;Preemption&quot; defined by the UHPTR valid bit set.</td>
</tr>
<tr>
<td>MI_ARB_CHECK</td>
<td>Move the head only if UHPTR valid bit is set.</td>
</tr>
</tbody>
</table>
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.

Note: 2-3 items above could happen during execution of a thread group so the HW may see the tail pointer updated before preemption starts.

Note: The driver needs to turn off RC6 during items 1 and 4.

**GPGPU_CSR_BASE_ADDRESS**
GPGPU Context Switch

The last GPGPU_WALKER or GPGPU_OBJECT in a command buffer should be followed by a MEDIA_STATE_FLUSH with the "Flush to GO" bit set to ensure that the last group can be preempted. The Interface Descriptor offset is limited to the range 0 to 31 when context switch is used.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW:GT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the GT1 SKU only, preemption can be used if the workload is split into chunks that will fit into the machine at one time. So each chunk must have 70 threads or less, 16 barriers or less, and 64k of SLM or less. A PIPE_CONTROL is used between each chunk to flush the pipe. Alternatively, preemption can be turned off for workloads that use SLM or barriers.</td>
<td></td>
</tr>
</tbody>
</table>
### 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:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Commands</th>
<th>Data Stored</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU</td>
<td>GPGPU_WALKER</td>
<td>CURBE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPGPU_OBJECT</td>
<td>CURBE</td>
<td></td>
</tr>
<tr>
<td>Media(Legacy)</td>
<td>Media_Object</td>
<td>CURBE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object</td>
<td>INDIRECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object</td>
<td>INLINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object</td>
<td>CURBE+INLINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object</td>
<td>CURBE+INDIRECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object</td>
<td>INLINE+INDIRECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object_Walker</td>
<td>CURBE+INLINE+INDIRECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object_Walker</td>
<td>CURBE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object_Walker</td>
<td>INLINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media_Object_Walker</td>
<td>CURBE+INLINE</td>
<td></td>
</tr>
</tbody>
</table>
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 [SNB+] > 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 RO header of the Thread Dispatch Payload for the GPGPU thread:

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.7</td>
<td>31:0</td>
<td><strong>Thread Group ID Z</strong>: This field identifies the Z component of the thread group that this thread belongs to.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.6</td>
<td>31:0</td>
<td><strong>Thread Group ID Y</strong>: This field identifies the Y component of the thread group that this thread belongs to.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Pointer</strong>: Specifies the 1K-byte aligned pointer to the scratch space. Format = GeneralStateOffset[31:10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td><strong>GPGPU Dispatch</strong>: Indicates that this dispatch is from the GPGPU pipe (see PIPELINE_SELECT command).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>FFTID</strong>: 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 = U8.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td><strong>Binding Table Pointer</strong>: 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]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td>Indicates the stack memory size. Range = [0,11] indicating [1K bytes, 2M bytes]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer</strong>: Specifies the 32-byte aligned pointer to the sampler state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>table.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = GeneralStateOffset[31:5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td><strong>Per Thread Scratch Space.</strong> 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.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format = U4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range = [0,10] indicating [2K bytes, 2M bytes] in powers of two.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td><strong>Barrier Enable:</strong> This field indicates that a barrier has been allocated for this kernel.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td><strong>SLM Enable:</strong> This field indicates that Shared Local Memory has been allocated for this kernel.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td><strong>BarrierID:</strong> This field indicates the barrier that this kernel is associated with.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format: U4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:10</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:4</td>
<td><strong>Interface Descriptor Offset.</strong> This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format = U6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:0</td>
<td><strong>Thread Group ID X:</strong> This field identifies the X component of the thread group that this thread belongs to.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:30</td>
<td><strong>Reserved:</strong> MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29:28</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td><strong>Shared Local Memory Index:</strong> Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4K memory block,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

829
<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16 possibilities cover the entire 64K shared memory per half-slice.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td>URB Handle: This is the URB handle indicating the URB space for use by the thread.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Project:</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>The GPGPU_OBJECT command is modified from the MEDIA_OBJECT command. It is no longer supported.</td>
<td></td>
</tr>
</tbody>
</table>

GPGPU_OBJECT
GPGPU_WALKER

GPGPU Mode [Project: IVB+, VLVT]

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 of them in order to select the execution masks correctly – see section *Execution Masks* on 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 |
| 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 7 6 5 4 3 2 1 0 11 10 9 8 7 6 5 4 |
| 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 |
| 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 |
| 11 10 9 8 7 6 5 4 3 2 1 0 11 10 9 8 |
| 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
| 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
```

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.

**Execution Masks**

The number of channels required by the GPGPU job may not evenly fit into the number of SIMD channels. That can leave some channels idle. The execution mask is used to tell the hardware which channels are to be used.

A thread group is modeled as a 3D solid with each channel acting as one X/Y/Z point in the solid. This can take the form of a line with 1024 channels with X from 0 to 1023 and constant Y/Z, a square with X=0 to 32 and Y=0 to 32, or a cube with X=0 to 9, Y=0 to 9, Z=0 to 9. Software needs to determine how these shapes are mapped onto the 32 SIMD32 channels per dispatch (or 16 SIM16, etc). The mapping per thread is assumed to be a 2D square of channels such as 8x4, 16x2, 32x1. Below is a diagram of a 22x6 thread group that is mapped onto a set of 8x4 SIMD32 channels:
Note that the dispatches to the top and left have execution masks of all-F, while all the right edge dispatches have the same execution mask; likewise all the bottom edge dispatches have the same execution mask. The bottom right is the logical-AND of the right and bottom edge dispatches.

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

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

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

The width and height counter would have a maximum of 1, and the depth counter would have a maximum of 2. The two dispatches would use the AND of the two masks, but since the bottom mask is F it would be the same as just the right mask.

The execution masks can also be used when the software wants to pack the channels rather than lay them out in a regular pattern:
In this case the width counter can have a maximum of 2, and the height and depth counters with a maximum of 1. The first dispatch will use the bottom mask only (all-F) and the second will use the right AND bottom mask to remove the channels that are not used.

Payload Storage

The MEDIA_CURBE_LOAD constant data is stored in the URB by CS and read out by TDL when the dispatch occurs. Data is sent in two sections – the cross-thread constant data (if present) is read out first by TDL from the CURBE handle in the transparent header, and the thread IDs are read out second via an inline handle. If cross-thread CURBE data is not present, then only the inline handle is used for the thread IDs. The inline payload with the execution counts is sent to VFE from CS. The execution counts are stored internal to VFE.

The thread id handle length is specified in the transparent header as the Push Constant Length for Buffer0. The cross-thread constant data handle length is specified in the transparent header as the Handle Length for Input Handles in DWord 0, bits 18:12.
Only 32 threads are allowed for SIMD32 to match the 1024 thread limit, requiring 32 execution count bytes, or 8 DW payload. SIMD16 and SIMD8 allow the full 64 thread per half-slice, and so require as much as 16 DWords.

The X/Y/Z payload size per dispatch is specified in the command, but a maximum size is 3 16-bit numbers per 1024 SIMD channels, or 6 kbytes.
The VFE manages the URB in GPGPU and generic/media modes.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW</td>
<td>The first 64 URB entries are reserved for the interface description, and CURBE data is placed after the IDs. URB handles are needed for indirect data and parent/child communication; when the VFE starts up it creates up to 128 handles by partitioning the remaining URB space into evenly spaced addresses and saving the resulting handles in a FIFO. The handles can then be treated just like ones created by the URBM - send to TD on dispatch and recovered on the handle return bus. MEDIA_VFE_STATE specifies the amount of CURBE space, the URB handle size and the number of URB handles. The driver must ensure that: $((\text{URB_handle_size} \times \text{URB_num_handle}) - \text{CURBE} - 64) \leq \text{URB_allocation_in_L3}$.</td>
</tr>
</tbody>
</table>
Thread Group Tracking

The TSG needs to keep track of the threads outstanding in a group to know when the thread group barrier and Shared Local Memory can be reclaimed. This can be done by keeping a counter per active thread group (up to 16 per half-slice) which increments when a new thread is sent out and decremented when the thread retires. The assigned barrier ID (with half-slice bit) is unique per thread group and much smaller than the thread group ID and so will be used to keep track of the thread group instead.

Since TSL sends the thread retirement via the Message Channel rather than the thread retirement bus, the barrier ID used to identify the thread group can be sent at the same time. A CAM will then match the ID with the counter to decrement.

There is a potential corner case of a thread group without barriers being partly dispatched, then retiring before the rest of the thread group is sent. This should be OK, since the lack of barriers means that there are no dependencies between threads.
Shared Local Memory Allocation

The Shared Local Memory is a 64k block per half-slice in the L3 that must be shared between all thread
groups on that half-slice. A new memory manager simular to the Scratch Space memory manager is used
to allocate this space.

We are only dispatching threads from a single Interface Descriptor at a time. If a new Interface Descriptor
is requested the pipe is drained and all shared memory recovered before starting to allocate new shared
memory. This means that only a single size of shared memory needs to be supported at once.

For simplicity, only power-of-2 sizes from 4k to 64k are allowed. The thread request will specify how
much is needed. The first thread of a Thread Group is marked as requiring a new shared local memory –
if not the old Shared Local Memory offset is sent with the dispatch.

A simple set of 16-bits is used to allocate 4k shared memory, with fewer bits used for larger sizes. A
priority encoder finds the first unused bit and the offset remembered as being associated with a
particular barrier id. The barrier id is then used to track the thread group.

When the Thread Group Tracking indicates that a thread group is completely retired, that section of
shared local memory can be reclaimed.
Software Managed Shared Local Memory

Software can optionally manage shared local memory. In this case, each thread command or thread group command will have the shared memory offset included – each command in a thread group must have the same offset, of 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

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>When the thread group tracker determines that a thread group has finished, the barrier becomes available to new thread groups.</td>
</tr>
</tbody>
</table>
Local Memory/Scratch Space

The Local Memory (not to be confused with Shared Local Memory, which is shared by all thread in a thread group) is allocated per thread dispatched to the EU.

The Scratch Space manager is used to provide between 1k and 2M bytes memory per thread.

The Scratch Space Manager automatically separates the scratch space for each subslice by 128 * Scratch_space_size. Memory usage can be optimized for large scratch spaces by adding an adjustment to the kernel to reduce this separate to the number of threads per subslice.

**New Scratch Space Pointer = Old Scratch Space Pointer - FFTID[8:7] (128 – 70) * Per Thread Scratch Space**

Bits [8:7] of the FFTID provide the subslice id; for each subslice we subtract the difference between the default separation between subslices and the minimum needed for a 10 EU and 7 threads per EU system.
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, which includes Media's functions, feature benefits, and how the features fit into graphics products as part of a whole solution. Media normally refers to products and services on digital computer-based systems that presents content, such as text, graphics, animation, video, audio, games, etc.

Media product features, as described in this PRM, include:

- Multi-format codec engine
- Video front end
- Media fixed functions
- Video encoding
- Video decoding
- Sampling

Media 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 and ISA is optimized for both.
• **Shared functions:** Include accelerators for filtered load, scatter, gather, and filter/blended store operations.
• **MFX:** 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.

### DevHSW:GT2 Media Pipelines

<table>
<thead>
<tr>
<th>Project</th>
<th>HSW:GT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additions/Changes for DevHSW:</td>
<td></td>
</tr>
<tr>
<td><strong>AES</strong>: Advanced Encryption Standard (Symmetric key encryption standard established by NIST).</td>
<td></td>
</tr>
<tr>
<td><strong>VEBOX</strong>: Relocated IECP from RC (Render Cache) and DN/DI from Sampler to VEBOX. This change essentially consolidated multiple shared functions to a dedicated fixed function HW, which improves performance and reduces power. Any video processing improvements can be made only inside VEBOX in the future.</td>
<td></td>
</tr>
<tr>
<td><strong>Improved VME</strong>: Integer Motion Estimation (IME) and Check and Refinement Engine (CRE). This is the third iteration of VME, which focuses on improving performance and flexibility vectors to translate to power and quality improvements. Among the HW changes are the IME repartitioning, HW accelerated chroma intra mode decision, HW assisted multi-reference support, and improved skip decision.</td>
<td></td>
</tr>
</tbody>
</table>
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.

- 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 assistant)

Terminologies

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVC</td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td>Child Thread</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EOB</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse Discrete Cosine Transform. It is the stage in the video decoding pipe between IQ and MC.</td>
<td></td>
</tr>
<tr>
<td>IQ</td>
<td>Inverse Quantization. It is a stage in the video decoding pipe between IS and IDCT.</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
<td>Security</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>IT</td>
<td>Inverse Integer Transform. It is the stage in AVC or VC1 video decoding pipe between IQ and MC.</td>
<td></td>
</tr>
<tr>
<td>MPEG</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td>Motion Compensation. It is part of the video decoding pipe.</td>
<td></td>
</tr>
<tr>
<td>MVFS</td>
<td>Motion Vector Field Selection – a four-bit field selecting reference fields for the motion vectors of the current macroblock.</td>
<td></td>
</tr>
<tr>
<td>PRT</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Parent Thread</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Root Thread</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Synchronized Root Thread</td>
<td>A root thread that is dispatched by TS upon a 'dispatch root thread' message.</td>
<td></td>
</tr>
<tr>
<td>TS</td>
<td>Thread Spawner. It is the second (and the last) fixed function in the general-purpose pipeline.</td>
<td></td>
</tr>
<tr>
<td>Unsynchronized Root Thread</td>
<td>A root thread that is automatically dispatched by TS.</td>
<td></td>
</tr>
<tr>
<td>VFE</td>
<td>Video Front End. It is the first fixed function in the general-purpose pipeline.</td>
<td></td>
</tr>
<tr>
<td>VLD</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
Hardware Feature Map in Products

The following table lists the hardware features in the media pipe.

## Video Front End Features in Device Hardware

<table>
<thead>
<tr>
<th>Features/Device</th>
<th>[DevSNB+]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Mode</td>
<td>Y</td>
</tr>
<tr>
<td>Root Threads</td>
<td>Y</td>
</tr>
<tr>
<td>Parent/Child Threads</td>
<td>Y</td>
</tr>
<tr>
<td>SRT (Synchronized Root Threads)</td>
<td>Y</td>
</tr>
<tr>
<td>PRT (Persistent Root Thread)</td>
<td>Y</td>
</tr>
<tr>
<td>Interface Descriptor Remapping</td>
<td>N</td>
</tr>
<tr>
<td>IS Mode (HW Inverse Scan)</td>
<td>N</td>
</tr>
<tr>
<td>VLD Mode (HW MPEG2 VLD)</td>
<td>N</td>
</tr>
<tr>
<td>AVC MC Mode</td>
<td>N</td>
</tr>
<tr>
<td>AVC IT Mode (HW AVC IT)</td>
<td>N</td>
</tr>
<tr>
<td>AVC ILDB Filter (in Data Port)</td>
<td>N</td>
</tr>
<tr>
<td>VC1 MC Mode</td>
<td>N</td>
</tr>
<tr>
<td>VC1 IT Mode (HW VC1 IT)</td>
<td>N</td>
</tr>
<tr>
<td>Stalling HW Scoreboard</td>
<td>Y</td>
</tr>
<tr>
<td>Non-stalling HW Scoreboard</td>
<td>Y</td>
</tr>
<tr>
<td>HW Walker</td>
<td>Y</td>
</tr>
<tr>
<td>HW Timer</td>
<td>Y</td>
</tr>
<tr>
<td>Features/Device</td>
<td>[DevSNB+]</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Pipelined State Flush</td>
<td>Y</td>
</tr>
<tr>
<td>HW Barrier</td>
<td>Y</td>
</tr>
</tbody>
</table>
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).
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.

GPGPU Media Pipe Differences

You can access the GPGPU pipe with the GPGPU_OBJECT and GPGPU_WALKER commands. A thread group id is associated with every dispatch, which is used to allocate and track barriers and Shared Local Memory. The GPGPU pipe has access to all the shared functions. The GPGPU pipe does not use the Scoreboard and should not dispatch child threads.

You can access the Media pipe with the various MEDIA_OBJECT* commands. Barriers and Shared Local Memory are not allocated for them. All shared functions are available. The Scoreboard is available to control dispatch depending on the completion of neighboring blocks.
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 [HSW]: 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

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.
### 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:

<table>
<thead>
<tr>
<th>GLOBAL LOOP PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Resolution</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCAL LOOP PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Resolution</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCAL INNER LOOP SPECIAL MODE SELECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Mode</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
</tbody>
</table>

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:

---

**Dual Mode of Local Loop**

The local Inner Loop Special mode selects are included to aid in the distribution of work, specifically with two slices in mind. Essentially, the local inner loop can be bisected and each half-walk can be directed inward towards the center of the image (dual). The local inner loop need not be bisected, and can either move away from the outer loop (repel) or move towards it (attract) when an even split is not desired:
In Dual mode, the sequence will alternate between two half-walks such that every-other output would go to the same slice. This effect will produce a more balanced workload to two slices as shown in the example below where the color of the block represents which slice it was dispatched to. This is the walker’s approach to fine-grained parallelism.

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 Outer Loop
        Global_Inner_Loop_Initiaization();
        While (Global_Inner_Loop_In_Bounds()){ //Global Inner Loop
            Local_Block_Boundary_Adjustment();
            Local_Outer_Loop_Initiaization();
            While (Local_Outer_Loop_In_Bounds()){ //Local Outer Loop
                Local_Middle_Loop_Initiaization();
                While (Local_Middle_Steps_Remaining()){ //Local Middle Loop
                    Local_Inner_Loop_Initiaization();
                    While (Local_Inner_Loop_Is_Shrinking()){ //Local Inner Loop
                        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
}

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
  o Current position \((x, y)\)
  o 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

Start → Load Inputs and Initialize

Global Outer Loop In Bounds?

Yes → Global Inner Loop Initialize

No → Calculate Next Global Outer (X,Y)

Global Inner Loop In Bounds?

Yes → Local Block Boundary Adjustment

No → Calculate Next Global Inner (X,Y)

Execute Local Loops

Stop

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

```
From Global Loops

Local Outer Loop In Bounds?
  No → Return to Global Loops
  Yes → Local Middle Loop Initialization

Local Middle Steps Remaining?
  No → Calculate Next Local Outer (X,Y)
  Yes → Local Inner Loop Initialization

Local inner Loop Shrinking?
  No → Calculate Next Local Middle (X,Y)
  Yes → Output
```
Scoreboard Control

A hardware mechanism controls the dispatch of root threads. Without using this hardware mechanism, only the dispatch of a SRT is managed by a parent root thread using the SRT message to TS.

There is a scoreboard hardware in TS unit. The scoreboard is addressed by the 18-bit (X, Y) scoreboard field in VFE DWord, where (X, Y) is typically used as the Cartesian coordinate of the working unit in a 2D frame but may be interpolated in other ways. When a root thread is dispatched, the entry at (X, Y) is marked. When the root thread is terminated, the corresponding bit in the scoreboard is cleared.

Each root thread may have up to eight dependencies. The dependency relation is described by the state value of Scoreboard Controls in terms of related distance of (deltaX, deltaY). There is a global scoreboard enabling in the state as well as the-per thread enabling for each dependency.

TS stalls the dispatch of a root thread if any scoreboard entry, which is denoted by (Scoreboard X + deltaX, Scoreboard Y + deltaY), matching with any enabled dependencies is marked as in-flight. The thread is dispatched only after all dependencies are cleared.

For a root thread, TS stalls the dispatch of the thread only if the dependent scoreboard entries of the thread are marked. It does not automatically stalls the dispatch for destination collision if (deltaX = 0, deltaY=0) is not set in the scoreboard state. This kind of scoreboard destination collision is due to the scoreboard wrap-around (or aliasing), which must be avoided. With 9-bit per X, Y field, the hardware scoreboard can support a frame that is subdivided up to 512x512 threads without a scoreboard aliasing.

In addition to the above stalling scoreboard, Media Pipe may also support a non-stalling scoreboard. With non-stalling, a thread is dispatched with the dependent threads marked. The thread dependency affects the issuing of a sendc instruction. See vol5d Execution Unit ISA for details.

Scoreboard Support in Device Hardware

<table>
<thead>
<tr>
<th>Device</th>
<th>Stalling scoreboard</th>
<th>Non-Stalling scoreboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DevSNB+]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Restrictions:

- 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 AVD 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 AVC-Style Dependency Example. For a macroblock in a MbAff pair, it may depend on up to
three, five or eight neighbors as shown in *AVC-Style Dependency Example* and *AVC-Style Dependency Example*, 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**

<table>
<thead>
<tr>
<th>D (x-1, y-1)</th>
<th>B (x, y+1)</th>
<th>C (x+1, y-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (x-1, y)</td>
<td>Current (x, y)</td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>D0 (x+1, 2y-2)</th>
<th>B0 (x, 2y-2)</th>
<th>C0 (x+1, 2y-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1 (x+1, 2y-1)</td>
<td>B1 (x, 2y-1)</td>
<td>C1 (x+1, 2y-1)</td>
</tr>
<tr>
<td>A0 (x-1, 2y)</td>
<td>Current (x, 2y)</td>
<td></td>
</tr>
<tr>
<td>A1 (x-1, 2y+1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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**

(a) Neighbors for the first macroblock in a ‘frame’ MbAff pair
(b) Neighbors for the second macroblock in a ‘frame’ MbAff pair

(b) Neighbors for the second macroblock in a ‘field’ MbAff pair

**Neighbor Availability**

<table>
<thead>
<tr>
<th>MbAff</th>
<th>FieldMbFlag</th>
<th>VertOrigin[0]</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>LB</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0/1</td>
<td>0/1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Progressive or Field picture</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1st Frame MbAff macroblock</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>✓</td>
<td>na</td>
<td>0</td>
<td>na</td>
<td>✓</td>
<td>2nd Frame MbAff macroblock</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1st Field MbAff macroblock</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2nd Field MbAff macroblock</td>
</tr>
</tbody>
</table>

**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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple Slice Considerations

For products with multiple slices such as DevHSW:GT3, the Render Cache is separate per slice with no hardware coherency. This means that the programmer must ensure coherency by one of these methods:

- Using write commit when writing to the Render Cache.
- Using Data Cache instead of the Render Cache.
- Different slices only access separate cache lines, using a hashing algorithm combined with the slice select bits of the MEDIA_OBJECT/GPGPU_OBJECT commands.

Interrupt Latency

Command Streamer is capable of context switching between primitive commands.

For all independent threads, it is not much a problem. The interrupt latency is dictated by the longest command that is likely to have the largest number of threads. For VLD mode, such a command may be corresponding to a largest slice in a high definition video frame. This is application dependent, there are not much host software can do. For Generic mode, programmer should consider to constrain the compute workload size of each thread.

In modes with child threads, a root thread may persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. The Software designer should partition the workload to restrict the duration of each root thread. For example, this may be achieved by partitioning a video frame and assigning separate primitive commands for different data partitions.

In modes with synchronized root threads, a synchronized root thread is dependent on a previous root or child thread. This means context switch is not allowed between the primitive command for the synchronized root thread and the one for the depending thread. So no command queue arbitration should be allowed between them. Software designer should also restrict the duration of such non-interruptible primitive command segments.
**Thread Spawner Unit**

The Thread Spawner (TS) unit is responsible for making thread requests (root and child) to the Thread Dispatcher, managing scratch memory, maintaining outstanding root thread counts, and monitoring the termination of threads.

**Thread Spawner block diagram**
Root Threads and Child Threads

Thread requests sourced from VFE are called root threads. These threads may be creating subsequent child threads.

Root Threads

A root thread may be a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode. Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.

Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
   - Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.

2. TS then builds the transparent header and the R0 header.

3. Finally, TS makes a thread request to the Thread Dispatcher.

4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.
   - It is the root thread’s responsibility (software) to guarantee that all its children have retired before the root thread can retire.

URB Handles

VFE is in charge of allocating URB handles for root threads. One URB handle is assigned to each root thread. The handle is used for the payload into the root thread.

Children Present is a command variable in the _OBJECT command.
If Children Present is not set (root-without-child case), TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched.

If Children Present is set (root-with-child case), the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.

**Root to Child Responsibilities**

Any thread created by another thread running in an EU is called a **child thread**. Child threads can create additional threads, all under the tree of a root which was requested via the VFE path.

A root thread is responsible of managing pre-allocated resources such as URB space and scratch space for its direct and indirect child threads. For example, a root thread may split its URB space into sections. It can use one section for delivering payload to one child thread as well as forwarding the section to the child thread to be used as return URB space. The child thread may further subdivide the URB section into subsections and use these subsections for its own child threads. Such process may be iterated. Similarly, a root thread may split its scratch memory 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**

![Diagram of thread relationships](image-url)
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.

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.

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2h</td>
<td>0h</td>
<td>00h</td>
<td>MEDIA_VFE_STATE</td>
</tr>
<tr>
<td>2h</td>
<td>0h</td>
<td>01h</td>
<td>MEDIA_CURBE_LOAD</td>
</tr>
<tr>
<td>2h</td>
<td>0h</td>
<td>02h</td>
<td>MEDIA_INTERFACE_DESCRIPTOR_LOAD</td>
</tr>
</tbody>
</table>
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**

**MEDIA_STATE_FLUSH**

<table>
<thead>
<tr>
<th>Project</th>
<th>Security</th>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

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**

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 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

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

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Pointer.</strong> Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:8</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------------------------------------------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td><strong>FFTID.</strong> This ID is assigned by TS and is a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unique identifier for the thread in comparison to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other concurrent root threads. It is used to free up</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>resources used by the thread upon thread completion.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td><strong>Binding Table Pointer.</strong> The 32-byte aligned</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pointer to the Binding Table. It is specified as an</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>offset from the <strong>Surface State Base Address.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = SurfaceStateOffset[31:5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer.</strong> Specifies the 32-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>aligned pointer to the sampler state table.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = GeneralStateOffset[31:5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td><strong>Per Thread Scratch Space.</strong> The amount of scratch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>space, in 1K-byte quantities, allowed to be used by</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>this thread. The value specifies the power that two</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is raised to, to determine the amount of scratch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>space.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0,11] indicating [1K bytes, 2M bytes] in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>powers of two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0.2</td>
<td>31:28</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:16</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:10</td>
<td></td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:4</td>
<td></td>
<td><strong>Interface Descriptor Offset.</strong> The offset from the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>interface descriptor base pointer to the interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptor that applies to this object, in units of</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>interface descriptors.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td><strong>Scoreboard Color</strong> (only with MEDIA_OBJECT_EX):</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field specifies which dependency color the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>current thread belongs to. It affects the dependency</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scoreboard control.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### DWord | Bits | Description | Project | Security
---|---|---|---|---
| | | Format = U4 | | |
| R0.1 | 31:28 | Reserved: MBZ | | |
| | 27:26 | Reserved: MBZ | | |
| | 25 | Reserved: MBZ | | |
| | 24:16 | **Scoreboard Y.**<br>This field provides the Y term of the scoreboard value of the current thread.<br>Format = U9 | | |
| | 15:12 | Reserved: MBZ | | |
| | 11:9 | Reserved: MBZ | | |
| | 8:0 | **Scoreboard X.**<br>This field provides the X term of the scoreboard value of the current thread.<br>Format = U9 | | |
| 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.<br>**Bit n (for n = 0...7):** Scoreboard n is dependent, where bit 24 maps to n = 0.<br>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

<table>
<thead>
<tr>
<th>GRF Register</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td><strong>R0 header</strong></td>
</tr>
<tr>
<td>R1 – R(m)</td>
<td>N/A</td>
<td><strong>Constants from CURBE when CURBE is enabled</strong></td>
</tr>
<tr>
<td>GRF Register</td>
<td>Example</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>$R(m+1)$</td>
<td>$R1$</td>
<td><strong>In-line Data block.</strong></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bit</th>
<th>Description</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.5</td>
<td>31:10</td>
<td><strong>Scratch Space Pointer.</strong> Specifies the 1K-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9:8</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>FFTID.</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>R0.4</td>
<td>31:5</td>
<td><strong>Binding Table Pointer:</strong> 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]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td>R0.3</td>
<td>31:5</td>
<td><strong>Sampler State Pointer.</strong> Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:0</td>
<td><strong>Per Thread Scratch Space.</strong> 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</td>
<td></td>
</tr>
<tr>
<td>R0.2</td>
<td>31:4</td>
<td><strong>Interface Descriptor Pointer.</strong> Specifies the 16-byte aligned pointer to <em>this thread’s</em> interface descriptor. Can be used as a base from which to offset child thread’s interface descriptor pointers. Format = GeneralStateOffset[31:4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3:1</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td><strong>Thread Restart Enable.</strong> If set, indicates that the persistent root thread</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bit</td>
<td>Description</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(PRT) is being restarted, and context should be restored from the context save area before executing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = Enable</td>
<td></td>
</tr>
<tr>
<td>R0.1</td>
<td>31:0</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td>R0.0</td>
<td>31:16</td>
<td>Reserved: MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td><strong>URB Handle.</strong> This is the URB handle indicating the URB space for use by the root thread and its children.</td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>GRF Register</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td><strong>R0 header</strong></td>
</tr>
<tr>
<td>R1 – R(m)</td>
<td>n/a</td>
<td><strong>Constants from CURBE when CURBE is enabled</strong></td>
</tr>
<tr>
<td>m is a non-negative value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(m+1)</td>
<td>R1</td>
<td><strong>In-line Data block</strong></td>
</tr>
</tbody>
</table>

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.

**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**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0.5</td>
<td>31:8</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7:0</td>
<td><strong>FFTID.</strong> 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 &quot;dereference resource&quot;, and is ignored by hardware otherwise.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.4</td>
<td>31:16</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15:10</td>
<td><strong>Dispatch URB Length.</strong> 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 &quot;peer&quot; 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 &quot;dereference resource&quot;. 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</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>command indirect/inline data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0,63] for child threads.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td><strong>URB Handle Offset.</strong> 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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is ignored if the <strong>Opcode</strong> is &quot;dereference resource&quot;.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format = U10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range = [0,1023]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.3</td>
<td>31:0</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.2</td>
<td></td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:28</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>BarrierID. This field indicates which one of the 16 Barriers this kernel is associated with.</td>
<td>Format: U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Ignored.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:10</td>
<td>Ignored.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:4</td>
<td>Interface Descriptor Offset. This field specifies the offset from the interface descriptor base pointer to the interface descriptor that is applied to this object. It is specified in units of interface descriptors.</td>
<td>Format = U6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Scoreboard Color (only with MEDIA_OBJECT_EX). This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control.</td>
<td>Format = U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.1</td>
<td>31:0</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0.0</td>
<td></td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:28</td>
<td>Ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>Shared Local Memory Index. Indicates the starting index for the shared local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
<td>Project</td>
<td>Security</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>Reserved: MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td>Dispatch URB Handle.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If Opcode (and Requester Type) is “spawn a child thread”: Specifies the URB handle for the child thread.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
EU Overview

The GEN instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Support for 3D graphics API (Application Programming Interface) Shader instructions is mostly native, meaning that GEN efficiently executes Shader programs. Depending on Shader program operation modes (for example, a Vertex Shader may be executed on a base of a vertex pair, while a Pixel Shader may be executed on a base of a 16-pixel group), translation from 3D graphics API Shader instruction streams into GEN native instructions may be required. In addition, there are many specific capabilities that accelerate media applications. The following feature list summarizes the GEN instruction set architecture:

- SIMD (single instruction multiple data) instructions. The maximum number of data elements per instruction depends on the data type.
- SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- Instruction level variable-width SIMD execution.
- Conditional SIMD execution via destination mask, predication, and execution mask.
- Instruction compaction.
- An instruction may executed in multiple cycles over a SIMD execution pipeline.
- Most GEN instructions have three operands. Some instructions have additional implied source or destination operands. Some instructions have explicit dual destinations.
- Region-based register addressing.
- Direct or indirect (indexed) register addressing.
- Scalar or vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

CoIssue/Dual Issue:

The Gen7 generation of EU allows two instructions to be issued at the same time (sometimes referred to as dual-issue or more generally co-issue). The two instructions issued are always from different threads. The terms FPU Pipe and EM Pipe are the terms used when referring to the two simultaneous pipes. The Gen7 implementation dual-issue capability is limited to only the most popular instructions and source operand modes. Later generations of EU expand on this concept to allow more operations.

Description:

- Opcodes: add, mov, mad, mul, cmp
- Datatype: single precision floats.
- 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 single precision float scalar is allowed.

  - Align16

  - Addressmode: Direct Addressing
  - Register File: GRF/NULL. No access to Accumulator.
  - Condition modifiers supported only for cmp.
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

![Diagram of AOS and SOA data structures]

GEN 3D and media applications take advantage of such broad architecture support and use both AOS and SOA data arrangements.

- Vertices in 3D Geometry (Vertex Shader and Geometry Shader) are arranged in AOS form and use SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA form and use SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA form, and occasionally in AOS form with possibly mixed modes of operation that uses region-based addressing extensively.

These are preferred methods; alternative arrangements may also be possible. Shared function resources provide data transpose capability to support both modes of operations: The sampler has a transpose for sample reads, the data port has a transpose for render cache writes, and the URB unit has a transpose for URB writes.

The following 3D graphics API Shader instruction is used in the following sections to illustrate various operation modes:

```plaintext
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 SIMDm\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:

\[
\text{add (4) } r3<4>.xyz:f \ r2<4>.yzwx:f \ r2.4<4>.zwxy:f \ \{\text{NoMask}\}
\]

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 operand. 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 \( \text{MaskCtrl} \) 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 sub-register 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**

![Diagram of SIMD4 Mode of Operation](image-url)
**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:

\[
\text{add (8) } r4<4>.xyz:f, r2<4>.yxzw:f, r3<4>.zwxy:f
\]

This instruction is subject to the execution mask, which initiated from the dispatch mask. If both program flows are available (e.g. Vertex Shader executed with two active vertices), the dispatch mask is set to 0x00FF. The operation of this GEN instruction is illustrated in *SIMD4x2 Mode of Operation* (a). The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 (doublewords 3 and 7) are unchanged due to the destination mask setting. If only one program flow is available (e.g. the same SIMD4x2 Vertex Shader with only one active vertex), the dispatch mask is set to 0x000F. The operation of the same instruction is shown in *SIMD4x2 Mode of Operation* (b).

**SIMD4x2 Examples with Different Emasks**

The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2, which is shared by the two program flows. The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

\[
\text{add (8) } r4<4>.xyz:f, r2<0>.yzwx:f, r3<4>.zwxy:f
\]

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**
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:

\[
\begin{align*}
    \text{add (16)} & \quad r18<1>:f \quad r4<8;8,1>:f \quad r14<8;8,1>:f // \text{dst}.x = \text{src0}.y + \text{src1}.z \\
    \text{add (16)} & \quad r20<1>:f \quad r6<8;8,1>:f \quad r16<8;8,1>:f // \text{dst}.y = \text{src0}.z + \text{src1}.w \\
    \text{add (16)} & \quad r22<1>:f \quad r8<8;8,1>:f \quad r10<8;8,1>:f // \text{dst}.z = \text{src0}.w + \text{src1}.x
\end{align*}
\]

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

![Diagram showing register mappings and instructions](image)

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

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:

\[
\begin{align*}
\text{add (16)} & \quad r18<1>:f \quad r2.1<0;1,0>:f \quad r14<8;8,1>:f \quad \{\text{Compr}\} \quad \text{// dst.x = src0.y + src1.z} \\
\text{add (16)} & \quad r20<1>:f \quad r2.2<0;1,0>:f \quad r16<8;8,1>:f \quad \{\text{Compr}\} \quad \text{// dst.y = src0.z + src1.w} \\
\text{add (16)} & \quad r22<1>:f \quad r2.3<0;1,0>:f \quad r10<8;8,1>:f \quad \{\text{Compr}\} \quad \text{// dst.z = src0.w + src1.x}
\end{align*}
\]

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 \ \{\text{Compr}\} \ \text{// dst.x = src0.y + src1.z}

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

\[
\begin{align*}
\text{add (8)} & \quad r18<1>:f \quad r4<8;8,1>:f \quad r14<8;8,1>:f \quad \text{// dst.x[7:0] = src0.y + src1.z} \\
\text{add (8)} & \quad r19<1>:f \quad r5<8;8,1>:f \quad r15<8;8,1>:f \quad \{\text{SecHalf}\} \quad \text{// dst.x[15:8] = src0.y + src1.z}
\end{align*}
\]

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.
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.

<table>
<thead>
<tr>
<th>Target Shared Functions supporting EOT messages</th>
<th>Target Shared Functions not supporting EOT messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null, DataPortWrite, URB, MessageGateway, ThreadSpawner</td>
<td>DataPortRead, Sampler</td>
</tr>
</tbody>
</table>

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.
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: Rn.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: Mn.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: Wn.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**

<table>
<thead>
<tr>
<th>Error</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad Shared Function ID</td>
<td>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.</td>
</tr>
<tr>
<td>Unknown opcode Incorrect message length</td>
<td>The destination shared function detects unknown opcodes (as specified in the send instructions &lt;desc&gt; field), and known opcodes where the message payload is either too long or too short, and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through and interrupt mechanism, then continues normal operation with the subsequent message.</td>
</tr>
<tr>
<td>Bad message contents in payload</td>
<td>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.</td>
</tr>
<tr>
<td>Incorrect response length</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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
<table>
<thead>
<tr>
<th>Error</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction is an immediate. Should a thread attempt to end</td>
<td>Should a thread attempt to end a thread using a <code>&lt;desc&gt;</code> sourced from a register, the EOT bit will not be recognized. In this case, the thread will</td>
</tr>
<tr>
<td>a thread using a <code>&lt;desc&gt;</code> sourced from a register, the EOT</td>
<td>continue to execute beyond the intended end of thread, resulting in a wide range of error conditions.</td>
</tr>
<tr>
<td>bit will not be recognized. In this case, the thread will</td>
<td></td>
</tr>
<tr>
<td>continue to execute beyond the intended end of thread,</td>
<td></td>
</tr>
<tr>
<td>resulting in a wide range of error conditions.</td>
<td></td>
</tr>
<tr>
<td>Two outstanding messages using overlapping GRF destinations</td>
<td>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.</td>
</tr>
<tr>
<td>ranges</td>
<td></td>
</tr>
</tbody>
</table>
 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

<table>
<thead>
<tr>
<th>Register File</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Register File (GRF)</td>
<td>r#</td>
<td>General purpose read write registers</td>
</tr>
</tbody>
</table>

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 registers, 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.
ARF Registers

ARF Registers Overview

Besides GRF registers that are directly indicated by 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 following table.

### Summary of Architecture Registers

<table>
<thead>
<tr>
<th>Register Type (RegNum [7:4])</th>
<th>Register Name</th>
<th>Register Count</th>
<th>Description</th>
<th>Project</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>null</td>
<td>1</td>
<td>Null register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001b</td>
<td>a0.#</td>
<td>1</td>
<td>Address register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010b</td>
<td>acc#</td>
<td>2</td>
<td>Accumulator register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011b</td>
<td>f#.#</td>
<td>2</td>
<td>Flag register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td>ce#</td>
<td>1</td>
<td>Channel Enable register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101b</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110b</td>
<td>sp</td>
<td>1</td>
<td>Stack Pointer Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111b</td>
<td>sr0.#</td>
<td>1</td>
<td>State register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000b</td>
<td>cr0.#</td>
<td>1</td>
<td>Control register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001b</td>
<td>n#</td>
<td>2</td>
<td>Notification Count register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010b</td>
<td>ip</td>
<td>1</td>
<td>Instruction Pointer register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011b</td>
<td>tdr</td>
<td>1</td>
<td>Thread Dependency register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100b</td>
<td>tm0</td>
<td>2</td>
<td>TimeStamp register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101b</td>
<td>fc#.#</td>
<td>39</td>
<td>Flow Control register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110b</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 sub-register number field, SubRegNum, in the instruction word has 5 bits. It is used to address sub-register regions for an architecture register supporting register subdivision. The SubRegNum field is in units of bytes. Therefore, the maximum number of bytes of an architecture register is limited to 32. Depending on the alignment restriction of a register type, only certain encodings of SubRegNum field apply for an architecture register. The detailed definitions are provided in the following sections.

<table>
<thead>
<tr>
<th>Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>In general an ARF register can be dst (destination) or src0 (source 0, first source)</td>
<td>HSW</td>
</tr>
</tbody>
</table>
operand) for an instruction. Depending on the register and the instruction, other restrictions may apply.

### Access Granularity

ARF registers may be accessed with sub-register granularity according to the descriptions below and following the same rule of region-based addressing for GRF. The machine code for register number and sub-register 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 sub-register 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 sub-register being indicated as reserved). Writes to unpopulated sub-registers are dropped; there are no side effect. Reads from unpopulated sub-registers, 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4])</td>
<td>0000b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>1</td>
</tr>
<tr>
<td><strong>Default Value:</strong></td>
<td>N/A</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>N/A</td>
</tr>
<tr>
<td>Elements:</td>
<td>N/A</td>
</tr>
<tr>
<td>Element Size:</td>
<td>N/A</td>
</tr>
<tr>
<td>Element Type:</td>
<td>N/A</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>N/A</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>N/A</td>
</tr>
<tr>
<td>SecHalf Control?:</td>
<td>N/A</td>
</tr>
<tr>
<td>Indexable?:</td>
<td>No</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4])</td>
<td>0001b</td>
</tr>
<tr>
<td>Number of Registers</td>
<td>1</td>
</tr>
<tr>
<td>Default Value</td>
<td>None</td>
</tr>
<tr>
<td>Normal Access</td>
<td>RW</td>
</tr>
<tr>
<td>Elements</td>
<td>8</td>
</tr>
<tr>
<td>Element Size</td>
<td>16 bits</td>
</tr>
<tr>
<td>Element Type</td>
<td>UW or UD</td>
</tr>
<tr>
<td>Access Granularity</td>
<td>Word</td>
</tr>
<tr>
<td>Write Mask Granularity</td>
<td>Word</td>
</tr>
<tr>
<td>SecHalf Control?</td>
<td>N/A</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>

There are eight address sub-registers forming an 8-element vector. Each address sub-register contains 16 bits. Address sub-registers can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the send instruction.

**Register and Subregister Numbers for Address Register**

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = a0</td>
<td>a0.0:uw</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>a0.1:uw</td>
</tr>
<tr>
<td>00000b = a0</td>
<td>a0.2:uw</td>
</tr>
</tbody>
</table>

When register a0 or sub-registers in a0 are used as the address register for register-indirect addressing, the address sub-registers must be accessed as unsigned word integers. Therefore, the sub-register number field must also be word-aligned.
<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011b = a0.3:uw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100b = a0.4:uw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101b = a0.5:uw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110b = a0.6:uw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111b = a0.7:uw</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>However, when register a0 or sub-registers 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.</td>
</tr>
</tbody>
</table>

### Address Register Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31:16</td>
<td><strong>Address sub-register a0.15:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>6</td>
<td>31:16</td>
<td><strong>Address sub-register a0.13:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>5</td>
<td>31:16</td>
<td><strong>Address sub-register a0.11:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>4</td>
<td>31:16</td>
<td><strong>Address sub-register a0.9:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>3</td>
<td>31:16</td>
<td><strong>Address sub-register a0.7:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>2</td>
<td>31:16</td>
<td><strong>Address sub-register a0.5:uw.</strong> Follows the same format as a0.3.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Follows the same format as a0.2.</td>
</tr>
<tr>
<td>1</td>
<td>31:16</td>
<td><strong>Address sub-register a0.3:uw.</strong> This field, with only the lower 12 bits populated can be used as an unsigned integer for register-indirect register addressing.</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Format: U12</td>
</tr>
<tr>
<td>0</td>
<td>31:16</td>
<td><strong>Address sub-register a0.1:uw.</strong> This field can be used for register-indirect register addressing or</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>serve as message descriptor for a <em>send</em> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <em>send</em> instruction, it provides the higher 16 bits of <em>&lt;desc&gt;</em>. Format: U12 or U16.</td>
</tr>
<tr>
<td>15:0</td>
<td>Address sub-register a0.0:uw. This field can be used for register-indirect register addressing or serve as message descriptor for a <em>send</em> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <em>send</em> instruction, it provides the lower 16 bits of <em>&lt;desc&gt;</em>. Format: U12 or U16.</td>
<td></td>
</tr>
</tbody>
</table>

When used as a source or destination operand, the address sub-registers can be accessed individually or as a group. In the following example, the first instruction moves 8 address sub-registers 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 sub-registers, and the fourth instruction replicates r1.4 as an unsigned word to the next 4 address sub-registers.

```
  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<8;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 sub-registers can be accessed individually or as a group. When accessed as a group, the address sub-registers must be group-aligned. For example, when two address sub-registers are used for register indirect addressing, they must be aligned to even address sub-registers. 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 sub-registers are used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address sub-registers are a0.2:uw and a0.3:uw. The two align to a DWord group in the address register. However, the two address sub-registers 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:** GEN ISA supports per channel indexing for a source operand. As there are only 8 sub-fields in the address register (to save hardware cost), the execution size of an instruction using per-channel indexing is limited to 8. Software may reload the address register and use compression control SecHalf to complete a 16-channel computation.

**Implementation restriction:** When used as the source operand *<desc>* for the *send* instruction, only the first dword sub-register 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.*
**Performance Note:** There is only one scoreboard for the whole address register. When a write to some sub-registers is in flight, hardware stalls any instruction writing to other sub-registers. Software may use the destination dependency control (NoDDChk, NoDDClr) to improve performance in this case. Similarly, when a write to some sub-registers is in flight, hardware stalls any instruction sourcing other sub-registers until the write retires.

**Accumulator Registers**

**Accumulator Registers Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>0010b</td>
<td>All</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>2</td>
<td>HSW</td>
</tr>
<tr>
<td>Default Value:</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RW</td>
<td>All</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Accumulator Registers</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are two accumulator registers, acc0 and acc1.</td>
<td>HSW</td>
</tr>
</tbody>
</table>

**Register and Subregister Numbers for Accumulator Registers**

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = acc0</td>
<td>Reserved: MBZ</td>
<td>HSW</td>
</tr>
<tr>
<td>0001b = acc1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All other encodings are reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>acc0 and acc1 are supported for single-precision Float (F) only. Use QtrCtrl of Q2 or Q4 to access acc1.</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HSW</td>
</tr>
</tbody>
</table>

Examples:

// Updates acc0 and acc1 because it is SIMD16:
add (16) r10:f r11:f r12:f {AccWrEn}

// Updates acc0 because it is SIMD8:
add (8) r10:f r11:f r12:f {AccWrEn}

// Updates acc1. Implicit access to acc1 using QtrCtrl:
add (8) r10:f r11:f r12:f {AccWrEn, Q2}

// Updates acc1 for Half Floats using QtrCtrl:
add (16) r10:hf r11:hf r12:hf {AccWrEn, H2}

- It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "add (8) acc1:f acc0:f"). The result of such an instruction is unpredictable.
- Swizzling is not allowed when an accumulator is used as an implicit source or an explicit source in an instruction.
- Reading accumulator content with a datatype different from the previous write will result in undeterministic values.
- For any DWord operation, including DWord multiply, accumulator can store up to 8 channels of data, with only acc0 supported.
- When an accumulator register is an explicit destination, it follows the rules of a destination register. If an accumulator is an explicit source operand, its register region must match that of the destination register with the exception(s) described below.

Implementation Precision Restriction: As there are only 64 bits per channel in DWord mode (D and UD), it is sufficient to store the multiplication result of two DWord operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The DWord multiplication result is then 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.

Implementation Precision Restriction: As there are only 33 bits per channel in Word mode (W and UW), it is sufficient to store the multiplication result of two Word operands with and without source modifier as the result is up to 33 bits. Integers are stored in accumulator in 2’s complement form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into a risk of overflowing. When overflow occurs, only modular addition can generate a correct result. But in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UD (FFFFFFFFh) with D (FFFFFFFFh) results in 1FFFFFFFEh. The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.
## Accumulator Channel Precision

<table>
<thead>
<tr>
<th>Project</th>
<th>Data Type</th>
<th>Accumulator Number</th>
<th>Number of Channels</th>
<th>Bits Per Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>DF</td>
<td>acc0</td>
<td>4</td>
<td>64</td>
<td>When accumulator is used for Double Float, it has the exact same precision as any GRF register.</td>
</tr>
<tr>
<td>HSW</td>
<td>F</td>
<td>acc0/acc1</td>
<td>8</td>
<td>32</td>
<td>When accumulator is used for Float, it has the exact same precision as any GRF register.</td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Not supported data type.</td>
</tr>
<tr>
<td>D (UD)</td>
<td></td>
<td>acc0</td>
<td>8</td>
<td>33/64</td>
<td>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 <code>mul</code>, <code>mach</code>, and <code>shr</code> (or <code>mov</code>).</td>
</tr>
<tr>
<td>W (UW)</td>
<td></td>
<td>acc0</td>
<td>16</td>
<td>33</td>
<td>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.</td>
</tr>
<tr>
<td>B (UB)</td>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Not supported data type.</td>
</tr>
</tbody>
</table>

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 is 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:
```
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

#### Flag Register

**Flag Register Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>0011b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>[HSW]: 2</td>
</tr>
<tr>
<td><strong>Default Value:</strong></td>
<td>None</td>
</tr>
<tr>
<td>Normal <strong>Access:</strong></td>
<td>RW</td>
</tr>
<tr>
<td>Elements:</td>
<td>[HSW]: 2</td>
</tr>
<tr>
<td>Element <strong>Size:</strong></td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>Word</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>Word</td>
</tr>
<tr>
<td>SecHalf Control?</td>
<td>Yes</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>

There are two flag registers, f0 and f1.

Each flag register contains two 16-bit sub-registers. 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 sub-registers.

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:
Updates flag sub-register 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 sub-register are unchanged.

Flag registers and sub-registers can also be explicit source or destination operands.

The sel instruction does not update flags.

**Note:** When branching instructions are predicated, branching is evaluated on all channels enabled at dispatch. This means, the appropriate number of flag register bits must be initialized or used in predication depending on the execution mask (EMask). Uninitialized flags may result in undesired branching. For example, if using DMask as EMask and if all 32 channels of DMask are enabled, a SIMD8 kernel must initialize unused flag bits so that predication on branching is evaluated correctly.

### Register and Subregister Numbers for Flag Register

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = f0:ud</td>
<td>00000b = fn.0:uw</td>
</tr>
<tr>
<td>0001b = f1:ud</td>
<td>00010b = fn.1:uw</td>
</tr>
<tr>
<td>Other encodings are reserved.</td>
<td>Other encodings are reserved.</td>
</tr>
</tbody>
</table>

Reference an entire flag register as f0:ud or f1:ud. Reference the flag sub-registers as f0.0:uw, f0.1:uw, f1.0:uw, and f1.1:uw.

### Channel Enable Register

#### Channel Enable Register Summary

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4])</td>
<td>0100b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>1</td>
</tr>
<tr>
<td>Default Value:</td>
<td>N/A</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RO</td>
</tr>
<tr>
<td>Elements:</td>
<td>1</td>
</tr>
<tr>
<td>Element Size:</td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>DWord</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>N/A</td>
</tr>
<tr>
<td>SecHalf Control?</td>
<td>No</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>
Register and Subregister Numbers for Channel Enable Register

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = ce</td>
<td>00000b = ce:ud</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

Channel Enable Register Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:0</td>
<td><strong>Channel Enable Register ce0.0:ud</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Format: U32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field contains 32 bits of Channel Enables or the Execution Mask for the current instruction.</td>
</tr>
</tbody>
</table>

SP Register

SP Register Summary

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4])</td>
<td>0110b</td>
</tr>
<tr>
<td>Number of Registers</td>
<td>1</td>
</tr>
<tr>
<td>Default Value</td>
<td>Provided by the Dispatcher</td>
</tr>
<tr>
<td>Normal Access</td>
<td>RW</td>
</tr>
<tr>
<td>Elements</td>
<td>2</td>
</tr>
<tr>
<td>Element Size</td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity</td>
<td>DWord</td>
</tr>
<tr>
<td>Write Mask Granularity</td>
<td>DWord</td>
</tr>
<tr>
<td>SecHalf Control</td>
<td>No</td>
</tr>
<tr>
<td>Indexable</td>
<td>No</td>
</tr>
</tbody>
</table>

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.
**SP Register Fields**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0     | 31:0 | **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.  
\[ sp = [\text{scratch space pointer}] + [\text{scratch space}] - 1 \] |
| 1     | 31:0 | **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 RO.  
\[ sp\_limit = [\text{scratch space pointer}] + [\text{stack space limit}] \] |

**State Register**

**State Register Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>0111b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>2</td>
</tr>
<tr>
<td>Default Value:</td>
<td>Provided by the Dispatcher</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RW</td>
</tr>
<tr>
<td>Elements:</td>
<td>4</td>
</tr>
<tr>
<td>Element Size:</td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>Byte</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>N/A</td>
</tr>
<tr>
<td>SecHalf Control?</td>
<td>No</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>
Register and Subregister Numbers for State Register

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = sr0</td>
<td>Valid encoding range:</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>00000b – 01100b</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td></td>
</tr>
</tbody>
</table>

State Register Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:28</td>
<td>Reserved. MBZ.</td>
</tr>
<tr>
<td></td>
<td>27:24</td>
<td>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.</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>Priority Class. This field, when set, indicates the thread belongs to the high priority class, which has higher scheduling priority over any thread with this field cleared. The priority field below determines the relative priority within the same priority class. This field is initialized by the thread dispatcher at thread dispatch time and stays unchanged throughout the life span of the thread. This field is forwarded on the message bus to the message bus arbiter for all out-going messages. It serves as a priority hint for the target shared function. See the Shared Function chapters for whether and how a shared function uses this priority hint.</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>Reserved. MBZ.</td>
</tr>
<tr>
<td>18:16</td>
<td></td>
<td>Priority. This field is the relative aging priority of the thread. This field indicates the 'age' of this thread relative to other threads within the EU. No two threads in the same EU can have the same priority number (independent of the priority class value). Within the same priority class, an older thread (with a larger priority number) has higher schedule priority over a younger thread. This field is set to zero at a thread's dispatch. During a thread's run time, this field may or may not be incremented when a new thread is dispatched to the same EU. It is only incremented when another thread's priority number is incremented and reaches the same value. For example, if currently there is a thread with priority 0 on an EU, then dispatching a new thread to that EU causes the old thread's priority number to increment to 1. However, if the active thread (assuming for simplicity that there is only one) on an EU has a priority number 1 (or 2 or 3), then dispatching a new thread to this EU does not change the old thread's priority number. As threads on an EU may terminate in arbitrary order, the exact number for a thread depends on the dynamic execution of threads.</td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>[15] Reserved. MBZ.</td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>7:3</td>
<td>Reserved. MBZ.</td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>TID (The thread identifier). Specifies the thread slot that the current thread is assigned to. This field is set at thread dispatch.</td>
<td></td>
</tr>
<tr>
<td>1 (sr0.1:ud)</td>
<td>31:23</td>
<td>FFTID (Fixed Function Thread ID). [H5W] There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.</td>
</tr>
<tr>
<td>22:12</td>
<td>Reserved. MBZ.</td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>SLM OffsetSLM Offset used by the thread. These bits are Write Only through SW and an attempt to read will result in unpredictable results. If SLM offset is used, then these bits have to be updated at the start of the thread itself.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved. MBZ.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>2 (sr0.2:ud)</td>
<td>31:0</td>
<td>Dispatch Mask (DMask). This 32-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register. Format: U32</td>
</tr>
<tr>
<td>3 (sr0.3:ud)</td>
<td>31:0</td>
<td>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</td>
</tr>
<tr>
<td>0 (sr1.0:ud)</td>
<td>31:0</td>
<td>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.</td>
</tr>
<tr>
<td>1 (sr1.1:ud)</td>
<td>31:0</td>
<td>Hardware Defined State Register. Same as sr1.0</td>
</tr>
<tr>
<td>2 (sr1.2:ud)</td>
<td>31:0</td>
<td>Hardware Defined State Register. Same as sr1.0</td>
</tr>
<tr>
<td>3 (sr1.3:ud)</td>
<td>31:0</td>
<td>Hardware Defined State Register. Same as sr1.0</td>
</tr>
</tbody>
</table>
The Control register is a read-write register. It contains four 32-bit sub-registers that can be accessed individually.

Subregister \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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.

\texttt{cr0.2:ud} contains the \textbf{Application IP (AIP)} indicating the current thread IP when an exception occurs. \texttt{cr0.3:ud} is reserved. Values written to this sub-register are dropped; the result of reading from this sub-register is unpredictable.

Fields in Control registers also reference a virtual register called \textbf{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.

Register and Subregister Numbers for Control Register

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = cr0</td>
<td>00000b = cr0.0:ud. It contains general thread control fields.</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>00100b = cr0.1:ud. It contains exception status and control.</td>
</tr>
<tr>
<td>01000b = cr0.2:ud. It contains AIP.</td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

Control Register Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td><strong>Master Exception State and Control.</strong> 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.</td>
</tr>
<tr>
<td>30:16</td>
<td><strong>Reserved.</strong> MBZ.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><strong>Breakpoint Suppress.</strong> This bit specifies whether breakpoint exception is suppressed or not. This bit is normally set by software and cleared by hardware. If Master Exception Status and Control bit is 1, this bit is ignored by hardware. If Master Exception Status and Control bit is 0 (i.e. not in System Routine) and Breakpoint is enabled: If this bit is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This bit is provided to prevent infinite loops of jumping to the System Routine on a breakpoint condition. The System Routine must set this bit (and also clear the corresponding status and control bit) before returning to the application thread. This bit has no effect when Breakpoint Enable bits are cleared. This bit is initialized to 0. 0 = Breakpoint exception is not suppressed.</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Breakpoint exception is suppressed.</td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved</td>
<td>MBZ.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[HSW]: <strong>Double Precision Denorm Mode.</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>5:4</td>
<td>[HSW]: <strong>Rounding Mode.</strong> 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)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>Vector Mask Enable (VME).</strong> 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)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Single Program Flow (SPF).</strong> Specifies whether the thread has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m &gt; 1). This bit affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This bit is initialized by the Thread Dispatch. 0: Multiple Program Flows 1: Single Program Flow <strong>Programming Restrictions:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><strong>Accumulator Disable.</strong> 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</td>
<td></td>
</tr>
</tbody>
</table>
### Description

being disabled. Setting this bit has no effect if the accumulator is the explicit destination operand for an instruction. This bit is initialized to 0.

0: Enable accumulator update.
1: Disable accumulator update.

**Usage Notes:**

This control bit is primarily designed for the System Routine. That routine is not expected to use the accumulator, though it may need to use instructions that implicitly update the accumulator. To use such instructions in the System Routine, but still preserve the accumulator contents on returning to the application kernel, the System Routine would either (a) save and restore the accumulator, or (b) prevent the accumulator from being unintentionally modified. This control bit has been added for the latter method.

Software has the option to limit the setting of this control bit to strictly within the System Routine. If, by convention, this bit is clear within application kernels, the System Routine can simply set the bit upon entry and clear it before returning control to the application kernel. This usage model would not necessarily require cr0.0 to be saved/ restored in the System Routine. However, if by convention application kernels are permitted to set this bit, then the System Routine is required to preserve the content of this bit.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>IEEE floating-point mode for the F (Float) type.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ALT (alternative) floating-point mode for the F (Float) type.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not be set by software or left set by the system routine to avoid confusion. This bit is initialized to 0.</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td><strong>Stack Overflow Exception Status.</strong> 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.</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td><strong>Force Exception Status and Control.</strong> This bit when set, indicates a Forced Exception. It is set when force exception on receiving the broadcast Force Exception Halt. This is enabled in TD_CTL (Refer to Debug Chapter). The System Routine should reset this bit before returning to an application routine. This bit is initialized to 0.</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td><strong>Context Save Status.</strong> 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.</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td><strong>Context Restore Status.</strong> 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.</td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td><strong>Reserved. MBZ.</strong></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td><strong>Breakpoint Enable.</strong> Specifies whether the breakpoint exception is enabled or not. This bit is initialized by the Thread Dispatcher. Format = ENABLED: 0: Disabled 1: Enabled</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td><strong>Software Exception Enable.</strong> 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</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td><strong>Illegal Opcode Exception Enable.</strong> 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.</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td><strong>Stack Overflow Exception Enable</strong>. 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.</td>
</tr>
<tr>
<td>10:0</td>
<td></td>
<td><strong>Reserved</strong>. MBZ.</td>
</tr>
<tr>
<td>2 (cr0.2:ud)</td>
<td>31:3</td>
<td><strong>Application IP (AIP)</strong>. 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 <strong>Master Exception State and Control</strong> 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 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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>AIP Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint</td>
<td>IP</td>
</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>
</tbody>
</table>

(1) External Halt exception is asynchronous and not associated with an instruction.

When the System Routine changes the Master Exception State and Control field from 1 to 0, hardware restores IP from this register. This field is writable allowing the returning IP to be altered after an exception is handled.

| 2:0 |      | **Reserved**. MBZ. |

### 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>1001b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>3</td>
</tr>
<tr>
<td><strong>Default Value:</strong></td>
<td>No</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RO (RW – Context save/restore only)</td>
</tr>
<tr>
<td>Elements:</td>
<td>3</td>
</tr>
<tr>
<td><strong>Element Size:</strong></td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>DWord</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>DWord</td>
</tr>
<tr>
<td>SecHalf Control?:</td>
<td>No</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>

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.

**Note:** The sub-register numbers for n0.0 and n0.2 are swapped on a write, i.e., a destination of n0.0 is required to update n0.2 and n0.2 is required to update n0.0.

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximum allowed number of concurrent threads in the system. Therefore, the range of the thread-to-thread notification count in n0, is larger than the maximum number of threads computed by EUID * TID. There is only one bit for the host-to-thread notification count in n1.

**Note:** When thread context save/restore is enabled, the host to thread communication using n1 is not supported.

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 sub-register causes the thread to stall, waiting for a notification signal from outside targeting the same sub-register. 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

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = n0</td>
<td>00000b = n0.0:ud</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>00100b = n0.1:ud</td>
</tr>
<tr>
<td></td>
<td>01000b = n0.2:ud</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

Notification Register 0 Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:16</td>
<td>Reserved. MBZ.</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Thread to Thread Notification Count.</strong> 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</td>
</tr>
</tbody>
</table>

Notification Register 1 Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:1</td>
<td>Reserved. MBZ.</td>
</tr>
</tbody>
</table>

Notification Register 2 Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:16</td>
<td>Reserved. MBZ.</td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td><strong>Thread to Thread Notification Count.</strong> 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</td>
</tr>
</tbody>
</table>

Format of the Notification Register
**IP Register**

**IP Register Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>1010b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>1</td>
</tr>
<tr>
<td><strong>Default Value:</strong></td>
<td>Provided by the Dispatcher</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RW</td>
</tr>
<tr>
<td>Elements:</td>
<td>1</td>
</tr>
<tr>
<td>Element Size:</td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>DWord</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>DWord</td>
</tr>
<tr>
<td>SecHalf Control?:</td>
<td>No</td>
</tr>
<tr>
<td>Indexable?:</td>
<td>No</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = ip</td>
<td>00000b = ip:ud</td>
</tr>
<tr>
<td>All other encodings are reserved.</td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

**IP Register Fields**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Subfield Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31:3</td>
<td><strong>Ip.</strong> Specifies the current instruction pointer. This pointer is relative to the General State Base Address.</td>
</tr>
<tr>
<td></td>
<td>2:0</td>
<td><strong>Reserved.</strong> MBZ.</td>
</tr>
</tbody>
</table>

**TDR Registers**

**TDR Registers Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>1011b</td>
</tr>
<tr>
<td>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>8</td>
</tr>
<tr>
<td>Default Value:</td>
<td>No</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RO/CW</td>
</tr>
<tr>
<td>Elements:</td>
<td>8</td>
</tr>
<tr>
<td>Element Size:</td>
<td>16 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UW</td>
</tr>
<tr>
<td>Access Granularity:</td>
<td>Word</td>
</tr>
<tr>
<td>Write Mask Granularity:</td>
<td>Word</td>
</tr>
<tr>
<td>SecHalf Control?</td>
<td>No</td>
</tr>
<tr>
<td>Indexable?</td>
<td>No</td>
</tr>
</tbody>
</table>

There are 8 thread dependency registers (tdr0.0:uw to tdr0.7:uw) used by HW for the `sendc` instruction. These registers are read-only and can be accessed in 16-bit granularity.

When accessed explicitly, each thread dependency register has FFTID in the lower 8 bits, bits 8 to 14 are forced to zero by HW. Bit 15 is the valid bit, which indicate whether the current thread has a dependency on the dependency thread stored in this thread dependency register.

The thread dependency registers are read only, the valids can only be set with a thread dispatch, and are reset by broadcasting end of thread messages after a thread retired. The FFTID’s can only be changed with a thread dispatch. Any write into any of the TDR registers will clear the valid bit for the particular TDR if the write enable is true, the FFTID portion is strictly read only.

### Register and Subregister Numbers for TDR Registers

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011b = tdr0</td>
<td>00000b = tdr0.0:uw</td>
</tr>
<tr>
<td></td>
<td>00010b = tdr0.1:uw</td>
</tr>
<tr>
<td></td>
<td>00100b = tdr0.2:uw</td>
</tr>
<tr>
<td></td>
<td>00110b = tdr0.3:uw</td>
</tr>
<tr>
<td></td>
<td>01000b = tdr0.4:uw</td>
</tr>
<tr>
<td></td>
<td>01010b = tdr0.5:uw</td>
</tr>
<tr>
<td></td>
<td>01100b = tdr0.6:uw</td>
</tr>
<tr>
<td></td>
<td>01110b = tdr0.7:uw</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
</tbody>
</table>

### TDR Registers Fields

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWord</td>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td><strong>Valid7.</strong> This field indicates whether the thread specified by FFTID7 is still in-flight.</td>
</tr>
<tr>
<td></td>
<td>30:25</td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td></td>
<td>24:16</td>
<td><strong>FFTID7.</strong> 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</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td><strong>Valid6.</strong> This field indicates whether the thread specified by FFTID6 is still in-flight.</td>
</tr>
<tr>
<td>14:9</td>
<td></td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td>8:0</td>
<td></td>
<td><strong>FFTID6.</strong> 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</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td><strong>Valid5.</strong> This field indicates whether the thread specified by FFTID5 is still in-flight.</td>
</tr>
<tr>
<td></td>
<td>30:25</td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td></td>
<td>24:16</td>
<td><strong>FFTID5.</strong> 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</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td><strong>Valid4.</strong> This field indicates whether the thread specified by FFTID4 is still in-flight.</td>
</tr>
<tr>
<td>14:9</td>
<td></td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td>8:0</td>
<td></td>
<td><strong>FFTID4.</strong> 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</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td><strong>Valid3.</strong> This field indicates whether the thread specified by FFTID3 is still in-flight.</td>
</tr>
<tr>
<td></td>
<td>30:25</td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td></td>
<td>24:16</td>
<td><strong>FFTID3.</strong> 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</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td><strong>Valid2.</strong> This field indicates whether the thread specified by FFTID2 is still in-flight.</td>
</tr>
<tr>
<td>14:9</td>
<td></td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
<tr>
<td>8:0</td>
<td></td>
<td><strong>FFTID2.</strong> 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</td>
</tr>
<tr>
<td>0</td>
<td>31</td>
<td><strong>Valid1.</strong> This field indicates whether the thread specified by FFTID1 is still in-flight.</td>
</tr>
<tr>
<td></td>
<td>30:25</td>
<td><strong>Reserved.</strong> MBZ</td>
</tr>
</tbody>
</table>
DWord | Bits | Description
--- | --- | ---
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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4]):</td>
<td>1100b</td>
</tr>
<tr>
<td>Number of Registers:</td>
<td>1</td>
</tr>
<tr>
<td>Default Value:</td>
<td>0h</td>
</tr>
<tr>
<td>Normal Access:</td>
<td>RO/RW</td>
</tr>
<tr>
<td>Elements:</td>
<td>3</td>
</tr>
<tr>
<td>Element Size:</td>
<td>32 bits</td>
</tr>
<tr>
<td>Element Type:</td>
<td>UD</td>
</tr>
</tbody>
</table>

Timestamp Register

This register is a low latency timestamp source, TM, available as part of a thread’s Architectural Register File (ARF). This is a is free running counter, 64b in size, and exposed to the ISA as individual 32b high TmHigh and low TmLow unsigned integer source operands. As part of the EU’s register space, access to the timestamp has a low and deterministic latency and therefore can be used for intra-kernel high resolution performance profiling.

The TM features provides a 1-bit indicator TmEvent which identifies the occurrence of a time-impacting event such as context switch or frequency change since the last time any portion of the Timestamp register value was read by that thread. Software that uses the Timestamp capability should check this bit to identify when a relative time calculation may be suspect. To properly use this additional information, the instrumentation code should operate on the Timestamp register value as a whole (i.e. as an 8 dword register) so that the 64b time and this 1b value are captured simultaneously, as opposed to 32b portions, to eliminate a the chance of missing a TmEvent that might occur between accesses to 32b portions of this register.
Note: 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**

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

**Performance Register Fields**

<table>
<thead>
<tr>
<th>DWord</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (tm0.0:ud)</td>
<td>31:0</td>
<td>TmLow. The lower 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32</td>
</tr>
<tr>
<td>1 tm0.1:ud</td>
<td>31:0</td>
<td>TmHigh. The upper 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32</td>
</tr>
<tr>
<td>2 tm0.2:ud</td>
<td>31:1</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>3 tm0.3 (pm0)</td>
<td>31:0</td>
<td>Undefined Format: U32</td>
</tr>
<tr>
<td>4 tm0.4:ud (tp0)</td>
<td>31:16</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Pause Counter. The pause duration. A non-zero value written to this register causes execution of the thread to halt for the corresponding number of clocks. Lower 5 bits are always zero and therefore, writing value less than 64 must not result in a pause</td>
</tr>
<tr>
<td></td>
<td>[15:10] – Reserved, must be written as zero; when read, returns zero.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4:0] – Reserved, must be zero.</td>
<td></td>
</tr>
</tbody>
</table>
Flow Control Registers

Flow Control Registers Summary

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF Register Type Encoding (RegNum[7:4])</td>
<td>1101b</td>
</tr>
<tr>
<td>Number of Registers</td>
<td>39</td>
</tr>
<tr>
<td>Default Value</td>
<td>None</td>
</tr>
<tr>
<td>Normal Access</td>
<td>RW*</td>
</tr>
</tbody>
</table>

Register and Subregister Numbers for Flow Control Registers

<table>
<thead>
<tr>
<th>RegNum[3:0]</th>
<th>SubRegNum[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b = fc0</td>
<td>00000b-11111b = fc0.0–fc0.31.</td>
</tr>
<tr>
<td>0001b = fc1</td>
<td>00000b = fc1.0. All other encodings are reserved.</td>
</tr>
<tr>
<td>0010b = fc2</td>
<td>00000b = fc2.0. All other encodings are reserved.</td>
</tr>
<tr>
<td>0011b = fc3</td>
<td>00000b = fc3.0. 00001b = fc3.1. 00010b = fc3.2. 00011b = fc3.3. All other encodings are reserved.</td>
</tr>
<tr>
<td>0100b = fc4</td>
<td>00000b = fc4.0. All other encodings are reserved.</td>
</tr>
</tbody>
</table>

These are special hardware registers used in handling flow control operations. These registers may be accessed ONLY in context save/restore operation using the SIP. These registers are accessed with the 'MOV' opcode. Use of any other opcode or access of these registers in non-context save/restore modes may result in undeterministic behaviour of hardware.

These registers are accessed as 256b registers. Parts of the 256b register may be redundant, depending on the hardware implementation of each register. The fields "RegNum" and "SubRegNum" are used together to address these registers.
Immediate

Two forms of immediate are provided as a source operand: scalar and vector.

The immediate field in a GEN instruction has 32 bits. For a word or an unsigned word 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.

For a scalar immediate, it can be of any of the specified numeric data types from a word to a dword. Byte and unsigned byte are not supported as the smallest internal type of the execution pipeline is word. These two numeric types are reserved for future extensions.

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 Data Type Section 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 Data Type Section for the description of the packing of vector floats to a dword.

Restriction: 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
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 \textbf{size} of the region, or the size of the operand. The number of execution channels is called the \textbf{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 \textbf{generalized 2-dimensional} (2D) arrays in row-major order. The first dimension is named the \textbf{horizontal} dimension (data elements within a row) and the second dimension is termed the \textbf{vertical} dimension (data elements in a column). Here, horizontal/vertical and row/column are just symbolic notations. When the GRF registers are viewed as a row-major 2D array of memory, such a notation normally matches well with the geometric locations of the data elements of an operand. However, as the register region is fully described by the parameters discussed below, the data elements of a register region may not form a regular rectangular shape. For example, Vertical Stride parameter is allowed to be smaller than Horizontal Stride, making the rows of a register region interleave with each other. It should also note that the meanings of horizontal/vertical here is different than that used for the flag control in Section \textit{Flag Register}.

Specifically, a region-based description of a source operand can take the following format

\[
\text{RegFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type}
\]
Parameters are as the follows.

- **Register Region Origin** (*RegFile*, *RegNum* and *SubRegNum*): This set of parameters, including the register file, *RegFile*, the register number, *RegNum*, and the sub-register number, *SubRegNum*, describes the register region origin, which is the location of the first data element of the operand. *RegNum* is in unit of 256-bit and *SubRegNum* is in unit of the data element size.

- **Width** (*Width*): *Width* specifies the number of data elements along the horizontal dimension, or the number of data elements of a row.

- **Horizontal Stride** (*HorzStride*): *HorzStride* specifies the step size between two adjacent data elements within a row. It is in unit of data element size, which is determined by the data element *Type*.

- **Vertical Stride** (*VertStride*): *VertStride* specifies the step size between two adjacent data elements along the vertical dimension (or the step size between two rows). It is again in unit of data element size, which is determined by the data element *Type*.

- **Data Element Type** (*Type*): *Type* specifies numeric data type (float, word, byte, etc.) of the data elements. All data elements within a region must have the same type.

In **GEN**, GRF and register files consist of a sequence of 256-bit registers. When viewing the register file (GRF for example) as a sequence of 256-bit aligned registers, *RegNum* field provides the register number, thus for the name. *SubRegNum* provides the sub-field addressing within a register. However, when viewing the register file as a byte addressable memory array, (*RegNum* and *SubRegNum*) is just a byte address within the register file with *SubRegNum* providing the lower 5 bits and *RegNum* providing the higher bits.

The execution size is specified for each instruction by the parameter *ExecSize*. The size of the vertical dimension is *ExecSize/Width*, based on the rule that the size of regions must equal to the execution size.

**Region Parameters** depicts the register region description. The example shows a register region of *r4.1<16;8,2>:w*, where the shaded fields denote the data elements in the region and the numbers in these fields are the execution channel assignments. The register region assumes that an *ExecSize* of 16 is set for the instruction. Each data element is a word (as noted by the type field :w). The origin of the region is at the second word of *r4*, denoted by *r4.1*. Each row of the region has 8 data elements (words) that are 2 data elements (words) apart. The distance between two rows is 16 words. Note that the region shown is for illustration purpose only. It does not represent a typical usage model nor a performance optimized mode.

**An example of a register region** (*r4.1<16;8,2>:w*) **with 16 elements**
Region Parameters shows another example where the rows are interleaved. The region, having word data elements, starts at location r5.0:w. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location 5.2:w. And there are 8 elements per row. VertStride, the distance between two rows, is only 1 word, which is less than HorzStride. Therefore, the first element of the second row (channel number 8) is at r5.1:w, just next to channel number 0. It is clear from the picture that the two rows are interleaved.

By varying the region parameters, reader may construct other configurations. The next section provides more details on the region-based register addressing. However, there are restrictions imposed by hardware implementation, which can be found in the later sections of this chapter.

A 16-element register region with interleaved rows (r5.0<1;8,2>:w)

Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.

// Input: Type: ub | b | uw | w | ud | d | f | v
//RegNum: In unit of 256-bit register
//SubRegNum: In unit of data element size
//ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels
int ElementSize = (Type==b||Type==ub) ? 1 : (Type==w|Type==uw) ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum<<5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
    int Offset = RowBase;
    for (int x=0; x<Width; x++) {
        Address [Channel++] = Offset;
        Offset += HorzStride*ElementSize;
    }
    RowBase += VertStride * ElementSize;
}

As HorzStride and VertStride are specified independently (note that VertStride might be smaller than or equal to HorzStride), the region may take various shapes from a replicated scalar, a replicated vector, a vector of replicated scalars, a sliding window, to a non-overlapped 2D array.

A region-based description of a destination operand can take the following simplified format

RegFile RegNum.SubRegNum<HorzStride>:type

The destination operand is only allowed to have a 1 dimensional region. The Register Region Origin and Type are the same as for a source operand. The total number of elements is given by ExecSize. However, only HorzStride is required to describe the 1D region, not VertStride and Width.

As a source register region may cross multiple physical GRF registers, an instruction with such source operands may take more than two execution cycles to gather source data elements for execution. The destination register region is restricted to be within a physical GRF register. In other words, destination scatter writes over multiple registers are not supported.
Region Addressing Modes

There are two different register addressing modes: Direct register addressing and register-indirect register addressing. Depending on the register region description, the register-indirect register addressing mode can be further divided into three usages: 1x1 index region where only the origin of register region is provided by the address register, Vx1 index region where the offset of each row of the register region is provided by an address register, VxH index region where the offset of each data element is provided by an address register.

Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Direct Register Addressing and Direct Register Addressing are two examples of direct register addressing.

For the example in Direct Register Addressing, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, Src0 and Src1, have 16 bytes. The destination operand, Dst, has 16 words. There are 8 elements in a row for Src0 and Src1. The vertical stride of 16 bytes for Src0 and Src1 indicates that the first element and the 9th element are 16 bytes apart in the register file. Note that Src0 falls into the 256-bit physical GRF register starting at r1.0, but Src1 crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have C = 3 + 9.

A region description example in direct register addressing

For the example in Direct Register Addressing, the sizes of areas of Src0 and Src1 are the same, but Src0 contains a vector of replicated scalars. With HorzStride = 0 and Width = 8, the first row of 8 elements in Src0 is a replication of the byte at r1.14. Comparing ExecSize of 16 to Width of 8 indicates that there is a
second row of 8 elements in \textit{Src0}. With VertStride = 16, the second row in \textit{Src0} is a replication of the byte at \textit{r1.20} (20 = 14 + 16). Effectively, the 16 data elements of \textit{Src0} are \{1,1,1,1,1,1,1,1,4,4,4,4,4,4,4,4\}.

\textbf{A region description example in direct register addressing with src0 as a vector of replicated scalars}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig.png}
\caption{Register-Indirect Register Addressing with a 1x1 Index Region}
\end{figure}

\textbf{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.

\textit{Register-Indirect Register Addressing with a 1x1 Index Region} depicts an example for this addressing mode. For example, the presence of a full region description \(<16;8,1>\) for \textit{Src0} indicates that only the origin of the region is provided by the address register \textit{a0.0}.

An example illustrating register-indirect register addressing mode with a 1x1 index region
Register-Indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, the horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided and the following contiguous address registers are for the following rows. The total number of address registers used is inferred from the parameters \( \text{ExecSize} \) and \( \text{Width} \).

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

An example is provided in *Register-Indirect Register Addressing with a Vx1 Index Region*. The assembly syntax notion of a register region without vertical stride, \(<4,1>\), corresponding to the special encoding of vertical stride of 0xF in the instruction word, indicates the VxH or Vx1 mode of indirect register addressing. In this case, the origin for each row of src0 is provided by the address register. As \( \text{ExecSize}/\text{Width} = 2 \), 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)
Register-Indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When Width of the region is 1, the number of address registers used equals ExecSize.

An example is provided in Register-Indirect Register Addressing with a VxH Index Region. The absent of vertical stride in the region description <1,0> with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As ExecSize/Width = 8, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.

An example illustrating register-indirect register addressing mode with a VxH index region (Src0).
Add (8) r9.0<1>:f  r[a0.0]<1,0>:f  r8.0<4,1>:f
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.

Table: Execution Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Word. 16-bit signed integer.</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Doubleword. 32-bit signed integer.</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Float. 32-bit single precision floating-point number.</td>
<td></td>
</tr>
<tr>
<td>DF</td>
<td>Double Float. 64-bit double precision floating-point number.</td>
<td></td>
</tr>
</tbody>
</table>

The following rules explain the conversion of multiple source operand types, possibly a mix of different types, to one common execution type:

- For floating-point sources, all source operands must have the same floating-point type, with the exceptions below
  - A two-source floating-point instruction can have Float as the src0 type and VF (Packed Restricted Float Vector) as the immediate src1 type.
- Mixing floating-point and integer source types is not allowed. Either all source types must be one floating-point type or all source types must be integer types.
- Unsigned integers are converted to signed integers.
- Byte (B) or Unsigned Byte (UB) values are converted to a Word or wider integer execution type.
- If source operands have different integer widths, use the widest width specified to choose the signed integer execution type.

Note that when the execution data type is an integer type, it is always a signed integer type. For integer execution types, extra precision is provided within the hardware, including the accumulators, so that conversions from unsigned to signed do not affect instruction correctness.
Register Region Restrictions

A register region is described as **packed** if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist.

The following register region rules apply to the GEN implementation.

1. **General Restrictions Based on Operand Types**
   There are these general restrictions based on operand types:
   
   1. Where \( n \) is the largest element size in bytes for any source or destination operand type, \( \text{ExecSize} \times n \) must be \( \leq 64 \).
   2. When the **Execution Data Type** is wider than the destination data type, the destination must be aligned as required by the wider execution data type and specify a \( \text{HorzStride} \) 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 \( \text{Dst.HorzStride} \) 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 \( \text{ExecSize} \), with these restrictions:
   
   1. \( \text{ExecSize} \) must be greater than or equal to \( \text{Width} \).
   2. If \( \text{ExecSize} = \text{Width} \) and \( \text{HorzStride} \neq 0 \), \( \text{VertStride} \) must be set to \( \text{Width} \times \text{HorzStride} \).
   3. If \( \text{ExecSize} = \text{Width} \) and \( \text{HorzStride} = 0 \), there is no restriction on \( \text{VertStride} \).
   4. If \( \text{Width} = 1 \), \( \text{HorzStride} \) must be 0 regardless of the values of \( \text{ExecSize} \) and \( \text{VertStride} \).
   5. If \( \text{ExecSize} = \text{Width} = 1 \), both \( \text{VertStride} \) and \( \text{HorzStride} \) must be 0.
   6. If \( \text{VertStride} = \text{HorzStride} = 0 \), \( \text{Width} \) must be 1 regardless of the value of \( \text{ExecSize} \).
   7. \( \text{Dst.HorzStride} \) must not be 0.
   8. \( \text{VertStride} \) must be used to cross GRF register boundaries. This rule implies that elements within a ‘\( \text{Width} \)’ cannot cross GRF boundaries.

A. **Region Alignment Rules for Direct Register Addressing**
   1. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.
   2. A destination cannot span more than 2 adjacent GRF registers.
   3. When 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:
      - a. The destination region is entirely contained in the lower OWord of a register.
      - b. The destination region is entirely contained in the upper OWord of a register.
      - c. The destination elements are evenly split between the two OWords of a register.
• When an instruction has a source region that spans two registers and the destination spans two registers, the destination elements must be evenly split between the two registers and each destination register must be entirely derived from one source register. **Note:** In such cases, the regioning parameters must ensure that the offset from the two source registers is the same.

The examples below illustrate the behavior of the cases permitted:

```
// Case (a) First 8 elements are from r12 to r10 and second from r13 to r11:
mov (16) r10.0<2>:w r12<16;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12<8;8,1>:w
mov (8) r11.0<2>:w r13<8;8,1>:w
```

```
// Case (b) First 8 elements from r12.8 to r10 and second from r13 to r11:
mov (16) r10.0<2>:w r12.8<16;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12.8<8;8,1>:w
mov (8) r11.0<2>:w r13.8<8;8,1>:w
```

The following examples indicate cases that are not allowed:

```
// Not allowed, because the source has 12 elements from r12 and 4 from r13:
mov (16) r10.0<2>:w r12.4<4;4,1>:w

// Not allowed, because the destination has 14 elements in r10 and 2 in r11:
mov (16) r10.2<1>:w r12<16;8,1>:w
```

• When destination spans two registers, the source MUST span two registers. The exception to the above rule:
  1. When source is scalar, the source registers are not incremented.
  2. When source is packed integer Word and destination is packed integer DWord, the source register is not incremented but the source sub register is incremented. Note: When lower 8 channels are disabled, the sub register of source1 operand is not incremented. If the lower 8 channels are expected to be disabled, say by predication, the instruction must be split into pair of simd8 operations.

The examples below illustrate the behavior of the cases permitted:

```
// Case (a) Scalar source:
mov (16) r10.0<2>:w r12.0<0;1,0>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12.0<0;1,0>:w
mov (8) r11.0<2>:w r12.0<0;1,0>:w
```

```
// Case (b) First 8 elements from r12 to r10 and second from r12.8 to r11:
mov (16) r10.0<1>:d r12<8;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<1>:d r12<8;8,1>:w
mov (8) r11.0<1>:d r12.8<8;8,1>:w
```

```
// Case (c) Example for Notes
add (16) r10.0<1>:d r12<8;8,1>:w r13<8;8,1>:w
// The above instruction must be split into
```
1. **Special Cases for Byte Operations**

   1. When the destination type is byte (UB or B) only a ‘raw move’ using the `mov` instruction supports a packed byte destination register region: $\text{Dst.HorzStride} = 1$ and $\text{Dst.DstType} = \text{(UB or B)}$. This packed byte destination register region is not allowed for any other instructions, including a ‘raw move’ using the `sel` instruction, because the `sel` instruction is based on Word or DWord wide execution channels.

   2. There is a relaxed alignment rule for byte destinations. When the destination type is byte (UB or B), destination data types can be aligned to either the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case the destination data bytes can be either all in the even byte locations or all in the odd byte locations. This rule has two implications illustrated by this example:

      // Example:
      mov (8) r10.0<2>:b r11.0<8;8,1>:w  
      mov (8) r10.1<2>:b r11.0<8;8,1>:w  

      // $\text{Dst.HorzStride}$ must be 2 in the above example so that the destination 
      // sub-registers 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.

2. **Special Requirements for Handling Double Precision Data Types**

   1. In Align1 mode, all regioning parameters must use the syntax of a pair of packed floats, including channel selects and channel enables.

      // Example:
      mov (8) r10.0.xyzw:df r11.0.xyzw:df  

      // The above instruction moves four double floats. The .x picks the 
      // low 32 bits and the .y picks the high 32 bits of the double float.

   2. In Align1 mode, all regioning parameters like stride, execution size, and width are in units of element size. However in Align16 mode, the channel selects and channel enables must always be used in pairs of packed floats, because these parameters are defined for DWord elements ONLY.

      // Example:
      mov (4) r10.0<1>:df r11.0<4;4,1>:df  

      // The above instruction moves four double floats.

3. **Regioning Rules for Register Indirect Addressing**

   1. When the execution size and destination regioning parameters require two registers, each register is pointed to by adjacent index registers.

      // 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
2. When the destination requires two registers and the sources are indirect, the sources must use 1x1 regioning mode. In addition, the sources must be assembled from GRF registers each accessed by adjacent index registers in 1x1 regioning modes. The data for each destination GRF register is entirely derived from one source register.

   // 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[a0.1]:f r[a0.3]:f r[a0.5]:f
   // Each access, source and destination, is a 1x1 regioning access.

   // Case (b):
   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[a0.1]:f r[a0.3]:f r[a0.5]<0;1,0>:f

3. Indirect addressing on src1 must be a 1x1 indexed region mode.
4. When a Vx1 or a VxH addressing mode is used on src0, the destination must use ONLY one register.
5. Indirect addressing on the destination must be a 1x1 indexed region mode.
6. Data elements referenced by a single index within a source region cannot cross a 256-bit register boundary.
7. The lower bits of the AddressImmediate must not overflow to change the register address. The lower 5 bits of Address Immediate when added to lower 5 bits of address register gives the sub-register offset. The upper bits of Address Immediate when added to upper bits of address register gives the register address. Any overflow from sub-register offset is dropped.

4. Special Restrictions
1. **Note:** [DevHSW:GT2:A]: SIMD16 is not allowed for three-source instructions.
2. When an instruction is SIMD32, the low 16 bits of the execution mask are applied for both halves of the SIMD32 instruction. If different execution mask channels are required, split the instruction into two SIMD16 instructions.
3. Instructions with condition modifiers must not use SIMD32.
4. All flow control (branching) instructions must use the Align1 access mode.
5. When using Align16 mode for conversion of data elements of different sizes, both source and destination must be one register each.
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**

**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.

The image below 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 sub-register 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 sub-register.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple sub-registers.
Generation of predication mask
No Predication

When PredCtrl field of a given instruction is set to 0 (*no predication*), it indicates that no predication is applied to this instruction. Effectively, the resulting PMask is all 1’s. This is shown by the 2:1 multiplexer B controlled by the Pred Enable signal in Predication. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.
Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag sub-register (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 sub-register are output to MUX A. In this case, a bit in the selected flag sub-register controls the conditional execution of the corresponding execution channel. Let the selected flag sub-register 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 sub-register) 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;
}
```
\[
\begin{align*}
b\text{Tmp} &= (\text{PredInv} == \text{TRUE}) \oplus \neg b\text{Tmp} : b\text{Tmp}; \\
\text{PMask}[ch] &= \text{PMask}[ch+1] = \text{PMask}[ch+2] = \text{PMask}[ch+3] = b\text{Tmp};
\end{align*}
\]

In **Align1** access mode, horizontal combination is based on AND combination \(\text{any}\#h\) and OR combination \(\text{all}\#h\) 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.

\[
\text{If (Access Mode == Align1)} \{
\text{Switch (PredCtrl)}} \{
\text{Case .any2h: groupSize = 2; } \langle \text{op} \rangle = |; \text{ break;}
\text{Case .all2h: groupSize = 2; } \langle \text{op} \rangle = \&; \text{ break;}
\text{Case .any4h: groupSize = 4; } \langle \text{op} \rangle = |; \text{ break;}
\text{Case .all4h: groupSize = 4; } \langle \text{op} \rangle = \&; \text{ break;}
\text{Case .any8h: groupSize = 8; } \langle \text{op} \rangle = |; \text{ break;}
\text{Case .all8h: groupSize = 8; } \langle \text{op} \rangle = \&; \text{ break;}
\text{Case .any16h: groupSize = 16; } \langle \text{op} \rangle = |; \text{ break;}
\text{Case .all16h: groupSize = 16; } \langle \text{op} \rangle = \&; \text{ break;}
\}
\text{For (ch = 0; ch < 16; ch += groupSize)} \{
\text{For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)}
\text{bTmp = bTmp } \langle \text{op} \rangle \ f0.\#[ch+inc];
\text{For (inc = 0; inc < groupSize; inc ++)}
\text{PMask}[ch+inc] = b\text{Tmp};
\}
\}
\]
Predication with vertical combination uses both flag sub-register as inputs. The AND or OR combination is across the sub-registers on a channel by channel basis. This is shown by the following pseudo-code.

```plaintext
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 occured in any of the computation of the current instruction, including clamping, re-normalizing, and format conversion.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **NS0** bit: The NaN Source 0 bit indicates whether src0 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **NS1** bit: The NaN Source 1 bit indicates whether src1 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is also set to 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.")

<table>
<thead>
<tr>
<th>Conditional Modifier</th>
<th>Meaning</th>
<th>Resulting Flag Value (for an execution channel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.e</td>
<td>Equal-to</td>
<td>`(pre_zero &amp; ! (NS0</td>
</tr>
<tr>
<td>.ne</td>
<td>Not-Equal-to</td>
<td>`! (pre_zero &amp; ! (NS0</td>
</tr>
<tr>
<td>.g</td>
<td>Greater-than</td>
<td>`(! pre_sign &amp; ! pre_zero &amp; ! (NS0</td>
</tr>
<tr>
<td>.ge</td>
<td>Greater-than-or-equal-to</td>
<td>`((! pre_sign</td>
</tr>
<tr>
<td>.l</td>
<td>Less-than</td>
<td>`(pre_sign &amp; !pre_zero &amp; ! (NS0</td>
</tr>
<tr>
<td>.le</td>
<td>Less-than-or-equal-to</td>
<td>`((pre_sign</td>
</tr>
</tbody>
</table>

Flag Generation for *cmpn* Instructions (The Supported Conditional Modifiers are ".ge", and ".l")

<table>
<thead>
<tr>
<th>Conditional Modifier</th>
<th>Meaning</th>
<th>Resulting Flag Value (for an execution channel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ge</td>
<td>Greater-than-or-equal-to</td>
<td>`(! pre_sign</td>
</tr>
<tr>
<td>.l</td>
<td>Less-than</td>
<td>`(pre_sign</td>
</tr>
</tbody>
</table>
Flag Generation for All Instructions Other than `cmp/cmpn` Instructions (The Supported Conditional Modifiers are `.e`, `.ne`, `.g`, `.ge`, `.l`, `.le`, `.o`, and `.u`.)

<table>
<thead>
<tr>
<th>Conditional Modifier</th>
<th>Meaning</th>
<th>Resulting Flag Value (for an execution channel)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.e</code></td>
<td>Equal-to</td>
<td><code>(post_zero &amp; ! NC)</code>. This conditional modifier tests whether the result is equal to zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If either source is NaN (i.e. NC is true), the flag is forced to false.</td>
</tr>
<tr>
<td><code>.ne</code></td>
<td>Not-Equal-to</td>
<td><code>!(post_zero &amp; ! NC)</code>. This conditional modifier test whether the result is not equal to zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It takes exactly the reverse polarity as modifier <code>.e</code>.</td>
</tr>
<tr>
<td><code>.g</code></td>
<td>Greater-than</td>
<td><code>!(post_sign &amp; ! post_zero &amp; ! NC)</code>. This conditional modifier tests whether result is greater than zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If either source is a NaN (i.e. NC is true), the flag is forced to false.</td>
</tr>
<tr>
<td><code>.ge</code></td>
<td>Greater-than-or-equal-to</td>
<td>`!(post_sign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If either source is a NaN (i.e. NC is true), the flag is forced to false.</td>
</tr>
<tr>
<td><code>.l</code></td>
<td>Less-than</td>
<td><code>(post_sign &amp; ! post_zero &amp; ! NC)</code>. This conditional modifier tests whether result is equal to zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If either source is a NaN (i.e. NC is true), the flag is forced to false.</td>
</tr>
<tr>
<td><code>.le</code></td>
<td>Less-than-or-equal-to</td>
<td>`!(post_sign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If either source is a NaN (i.e. NC is true), the flag is forced to false.</td>
</tr>
<tr>
<td><code>.o</code></td>
<td>Overflow</td>
<td>OF. This conditional modifier tests whether the computed result causes overflow – the computed result is outside the range of the destination data type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: The legacy condition modifier behavior is different from IEEE exception Overflow flag. For inf float to int conversion, <code>.o</code> will set the legacy Overflow flag, but IEEE exception Overflow flag won’t be set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other internal conditional signals are ignored.</td>
</tr>
<tr>
<td><code>.u</code></td>
<td>Unordered</td>
<td>NC. This conditional modifier tests whether the computed result is a NaN (unordered).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other internal conditional signals are ignored.</td>
</tr>
</tbody>
</table>
## 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 \( \{\text{NoDDClr}, \text{NoDDhk}\} \) 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 \textit{send} instruction, there is only one condition that a programmer may use the \( \{\text{NoDDClr}, \text{NoDDChk}\} \) capability.

- If none of the two instructions is \textit{send}, there CANNOT be any destination hazard. This is because instructions within a thread are dispatched in order (single-issued) and the execution pipeline is in-order and has a fixed latency.

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 filed with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand src0. In this case, the second source operand src1 must be set to null and also with the same type as the first source operand src0. It is a special case when src0 is an immediate, as an immediate src0 uses DW3 of the instruction word, which is normally used by src1. In this case, src1 must be programmed with register file ARF and the same data type as src0.
**Instruction Prefetch**

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 instructions (128 bytes) beyond the end of the kernel program – possibly into the next memory page. Although these instructions will not be executed, software must account for the prefetch in order to avoid invalid page access faults. One possible (though inefficient) solution would be to pad the end of all kernel programs with 8 NOOP instructions. A more efficient approach would be to ensure that the page after all kernel programs is at least valid (even if mapped to a dummy page). Note that the **General State Access Upper Bound** field of the STATE_BASE_ADDRESS command can be used to prevent memory accesses past the end of the General State heap (where kernel programs must reside).
ISA Introduction

This chapter contains these sections that introduce this volume.

- Introducing the Execution Unit
- EU Terms and Acronyms
- EU Changes by Processor Generation
- EU Notation

Subsequent chapters cover:

- EU Data Types
- Execution Environment
- Exceptions
- Instruction Set Summary
- Instruction Set Reference
- EU Programming Guide

The EU Programming Guide provides some useful examples and information but is not a complete or comprehensive programming guide.
Introducing the Execution Unit

This section introduces the Execution Unit (EU), a simple and capable processor within the GPU that supports graphics processing within the graphics pipelines, can do general purpose computing (GPGPU), and responds to exceptional conditions via the System Routine.

The EU provides parallelism at two levels: thread and data element. Multiple threads can execute on the EU; the number executing concurrently depends on the processor and is transparent to EU code. Each thread has its own registers (GRF and ARF, described below). Most EU instructions operate on arrays of data elements; the number of data elements is normally the ExecSize (execution size) or number of channels for the instruction. A channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels is independent of the number of physical ALUs or FPUs for a particular graphics processor.

EU native instructions are 128 bits (16 bytes) wide. Some combinations of instruction options can use compact instruction formats that are 64 bits (8 bytes) wide. Identifying instructions that can be compacted and creating the compact representations is done by software tools, including compilers and assemblers.

Data manipulation instructions have a destination operand (dst) and one, two, or three source operands (src0, src1, or src2). The instruction opcode determines the number of source operands. An instruction’s last source operand can be an immediate value rather than a register.

Data read or written by a thread is generally in the thread’s GRF (General Register File), 128 general registers, each 32 bytes. A data element address within the GRF is denoted by a register number (r0 to r127) and a sub-register number. In the instruction syntax, sub-register numbers are in units of data element size. For example, a :d (Signed Doubleword Integer) element can be in sub-register 0 to 7, corresponding to byte numbers in the instruction encoding of 0, 4, ... 28.

Note: 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 sub-register values using boolean formulas determined by the PredCtrl (Predicate Control) and PredInv (Predicate Inversion) instruction fields. See the Predication section.

- **Masking** is the overall process of determining which channels execute for a given instruction based on five factors:
  - Number of channels (only channels in [0, ExecSize - 1] can execute)
  - Execution mask (EMask)
  - Whether the channel is on the instruction (if not in Single Program Flow mode and MaskCtrl is not NoMask)
  - Predicate mask (PMask)
  - In Align16 mode, any enabling of channels using the Dst.ChanEn instruction field (if MaskCtrl is not NoMask).

- **Regioning** specifies an array of data elements contained in one or two registers, with options for scattering, interleaving, or repeating data elements in registers using width and stride values, subject to significant constraints. Regioning also includes access mode (Align1 or Align16) and addressing mode (Direct or Indirect). See the Registers and Register Regions section.

- **Swizzling** allows small scale reordering of data elements within groups of four at the input using the modulo 4 channel names x, y, z, and w. For example, a swizzle of .wzyx with an ExecSize of 8 reads execution channels 0 to 7 from these input channels: 3, 2, 1, 0, 7, 6, 5, and 4. Swizzling is only available in the Align16 access mode, described in the Execution Environment chapter.

- **Type Conversions** do any needed conversion from source data type to execution data type and from execution data type to destination data type. See Execution Data Type for more information. Each instruction description indicates what combinations of data types are supported.

- **Source Modification** modifies a source operand just before doing the requested operation. For a numeric operation, the choices are:
  - No modification (normal).
  - - indicating negation.
  - (abs) indicating absolute value.
  - -(abs) indicating a forced negative value.
Source modification logically occurs after any conversion from source data type to execution data type. Each instruction description indicates whether it supports source modification.

- **Saturation** clamps result values to the nearest value within a saturation range determined by the destination type. For a floating-point type, the saturation range is [0.0, 1.0]. For an integer type, the saturation range is the entire range for that type, for example [0, 65535] for the UW (Unsigned Word) type. Each instruction description indicates whether it supports saturation.
• **Accumulator Updates** optionally update the accumulator register or registers in the ARF with destination values as a side effect of instruction execution. The AccWrCtrl instruction field enables accumulator updates. The Accumulator Disable flag in control register 0 (cr0) can be used to disable accumulator updates, regardless of AccWrCtrl values; for example, this flag may be used in the System Routine.

• **Flag Updates** optionally update a flags register and sub-register (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.

**Note:** 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<8,1>:f r4.0<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

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT mode</td>
<td>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.</td>
</tr>
<tr>
<td>ALU</td>
<td>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.</td>
</tr>
<tr>
<td>AOS</td>
<td>Array Of Structures. Also see <a href="#">SOA</a>.</td>
</tr>
<tr>
<td>ARF</td>
<td>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.</td>
</tr>
<tr>
<td>byte</td>
<td>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.</td>
</tr>
<tr>
<td>channel</td>
<td>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.</td>
</tr>
<tr>
<td>channel</td>
<td>Supports up to 32 channels.</td>
</tr>
<tr>
<td>compact instruction</td>
<td>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.</td>
</tr>
<tr>
<td>compressed instruction</td>
<td>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.</td>
</tr>
<tr>
<td>denorm</td>
<td>A very small but nonzero number in IEEE mode, with a magnitude less than the smallest normalized floating-point number representable in a particular floating-point format. Denormals lose precision as their values approach zero, called gradual underflow.</td>
</tr>
<tr>
<td>DWord</td>
<td>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.</td>
</tr>
<tr>
<td>EOT</td>
<td>End of Thread. A flag set on a send or sendc instruction to terminate a thread's execution on the EU.</td>
</tr>
<tr>
<td>EU</td>
<td>Execution Unit. The single GPU unit described in this volume. This volume describes individual data parallel execution paths within a thread in the EU as channels. A few fields, like EUID, use EU to refer to a particular hardware resource used to implement the overall EU.</td>
</tr>
<tr>
<td>exception</td>
<td>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.</td>
</tr>
<tr>
<td>ExecSize</td>
<td>The number of execution channels for a particular instruction. Channels within that number are enabled or disabled by various masks.</td>
</tr>
<tr>
<td>floating-point</td>
<td>Numeric types that allow fractional values and often a wider range than integer types. The EU</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>supports</td>
<td>binary floating-point types including the single precision type and the double precision type defined by the IEEE 754 standard.</td>
</tr>
<tr>
<td>GEN</td>
<td>GEN is sometimes used to refer to Intel's mainstream GPU architecture integrated with recent CPU generations.</td>
</tr>
<tr>
<td>GRF</td>
<td>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 sub-register number, or indirectly addressed using an address sub-register (index register) and an address immediate offset.</td>
</tr>
<tr>
<td>IEEE mode</td>
<td>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.</td>
</tr>
<tr>
<td>index register</td>
<td>An address sub-register when used for indirect addressing.</td>
</tr>
<tr>
<td>inf</td>
<td>Infinity, +inf or -inf, as a floating-point value in IEEE mode.</td>
</tr>
<tr>
<td>instruction</td>
<td>In this volume, instruction always refers to an EU instruction.</td>
</tr>
<tr>
<td>ISA</td>
<td>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, FPs, 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.</td>
</tr>
<tr>
<td>LSB</td>
<td>Least significant bit.</td>
</tr>
<tr>
<td>message</td>
<td>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.</td>
</tr>
<tr>
<td>MSB</td>
<td>Most significant bit.</td>
</tr>
<tr>
<td>NaN</td>
<td>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.</td>
</tr>
<tr>
<td>native instruction</td>
<td>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.</td>
</tr>
<tr>
<td>OWord</td>
<td>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.</td>
</tr>
<tr>
<td>packed</td>
<td>A register region is described as packed if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist. The immediate vector data types are all described as Packed because each such type packs several small data elements into a 32-bit immediate value.</td>
</tr>
</tbody>
</table>
## EU Numeric Data Types (Listed Alphabetically by Short Name)

<table>
<thead>
<tr>
<th>Short Name</th>
<th>Assembler Syntax</th>
<th>Long Name</th>
<th>Size in Bytes</th>
<th>Size in Bits</th>
<th>Integral or Float</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>:b</td>
<td>Signed Byte Integer</td>
<td>1</td>
<td>8</td>
<td>I</td>
<td>Signed integer in the range -128 to 127.</td>
</tr>
<tr>
<td>D</td>
<td>:d</td>
<td>Signed Doubleword Integer</td>
<td>4</td>
<td>32</td>
<td>I</td>
<td>Signed integer in the range -$2^{31}$ to $2^{31}$ - 1.</td>
</tr>
<tr>
<td>DF</td>
<td>:df</td>
<td>Double Float</td>
<td>8</td>
<td>64</td>
<td>F</td>
<td>Double precision floating-point number.</td>
</tr>
<tr>
<td>F</td>
<td>:f</td>
<td>Float</td>
<td>4</td>
<td>32</td>
<td>F</td>
<td>Single precision floating-point number.</td>
</tr>
<tr>
<td>UB</td>
<td>:ub</td>
<td>Unsigned Byte Integer</td>
<td>1</td>
<td>8</td>
<td>I</td>
<td>Unsigned integer in the range 0 to 255.</td>
</tr>
<tr>
<td>UD</td>
<td>:ud</td>
<td>Unsigned Doubleword Integer</td>
<td>4</td>
<td>32</td>
<td>I</td>
<td>Unsigned integer in the range 0 to $2^{32}$ - 1.</td>
</tr>
</tbody>
</table>

The next table lists all EU numeric data types. See the [Numeric Data Types](#) section for more information about each data type.
### Short Name | Assembler Syntax | Long Name | Size in Bytes | Size in Bits | Integral or Float | Description
--- | --- | --- | --- | --- | --- | ---
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)

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulators</td>
<td>acc0, acc1</td>
<td>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 <code>mac</code> (Multiply Accumulate), use the accumulators as an implicit source operand, useful for some iterative calculations.</td>
</tr>
<tr>
<td>Address Register</td>
<td>a0.s</td>
<td>Holds sub-registers primarily used for indirect addressing. Each sub-register is a 16-bit UW (Unsigned Word) value. For an indirectly addressed operand or element, the sub-register value plus an AddImm signed offset field determines the byte address (RegNum and SubRegNum) within the register file (GRF ). There are 8 address sub-registers.</td>
</tr>
<tr>
<td>Control Register</td>
<td>cr0.s</td>
<td>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.</td>
</tr>
<tr>
<td>Flags</td>
<td>fr.s</td>
<td>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 sub-registers.</td>
</tr>
</tbody>
</table>
| 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
<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Register</td>
<td>null</td>
<td>Indicates a non-existent operand. Unused operands in the instruction format, like the unused second source operand field in a <code>mov</code> 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.</td>
</tr>
<tr>
<td>State Register</td>
<td>sr0.s</td>
<td>Contains thread identification and scheduling fields, and mask fields for enabling or disabling channels.</td>
</tr>
</tbody>
</table>

**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

- thread-to-thread communication (see *Message Gateway, Media*)
- synchronization of thread output to memory buffers (see *Geometry Shader*).

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs.
EU Changes by Processor Generation

This section describes how the EU changes for particular processor generations. Instruction compaction tables can differ for each generation, so that is not mentioned in these lists. Particular readers and audiences can see only certain content in this section. Notes and 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.

Pre-Haswell

These features or behaviors are added Pre-Haswell, continuing for Haswell:

- The maximum ExecSize increases to 32, for byte or word operands.
- Increase the number of flag registers from one to two.
- Add the NibCtrl field, used with QtrCtrl to select groups of channels or flags.
- Add the DF (Double Float) data type, the first time an 8-byte data type is supported. DF only supports the IEEE floating-point mode and not the ALT floating-point mode.
- Add a shared source data type field and a destination data type field for instructions with three source operands, allowing F (Float), DF (Double Float), D (Signed Doubleword Integer), or UD (Unsigned Doubleword Integer) types to be specified.
- Add bit manipulation instructions: bfi1, bfi2, bfrev, cbit, fbh, and fbl.
- Add the integer addc (Add with Carry) and subb (Subtract with Borrow) instructions.
- Add the brc (Branch Converging) and brd (Branch Diverging) instructions.
- For the cmp and cmpn instructions, relax the accumulator restrictions.
- For the sel instruction, remove the accumulator restriction.
- Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0.

Haswell

- DF (Double Float) operands use an element size of 8. Regioning and channel parameters for the DF type are determined normally, in the same way as for other types.
- Add the channel enable register, flow control registers, and stack pointer register in the ARF.
- In the Control Register, add the Force Exception Status and Control, Context Save Status, and Context Restore Status bits.
- Relative instruction offsets (JIP, UIP) are now 32-bit values in units of bytes (rather than 16-bit values using 8-byte units) for some instructions: brc, brd, call, and jmpi.
- A call instruction can get the relative instruction offset (JIP) from a register.
- Add the calla (Call Absolute) instruction.
- A mov instruction with different source and destination types can now use conditional modifiers.

These features or behaviors are specific to HSW and may not continue to later generations:

- Add the dim (Double Precision Floating Point Immediate Data Move) instruction.
• The *f16to32* and *f32to16* instructions are supported to convert between half-precision float and Float.
• The *mul* instruction limits integer multiplication involving DWords so that only the low 16 bits of src1 are used even if src1 is a DWord.
• The *sel* (Select) instruction does not support an *ExecSize* of 32.
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, Note/Notes, 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 are in plain text in square brackets. For example, [HSW].

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.
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
  - Comparison of Floating Point Numbers
  - Min/Max of Floating Point Numbers
- Alternative Floating Point Mode

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**

```
  7  0
```

**B: Byte, 8-bit Signed Integer**

```
  7  6  0
  5
```
### Execution Unit Integer Data Types

<table>
<thead>
<tr>
<th>Notation</th>
<th>Size in Bits</th>
<th>Name</th>
<th>Range</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UB</td>
<td>8</td>
<td>Unsigned Byte Integer</td>
<td>[0, 255]</td>
<td>DevSNB+</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>Signed Byte Integer</td>
<td>[-128, 127]</td>
<td>DevSNB+</td>
</tr>
<tr>
<td>UW</td>
<td>16</td>
<td>Unsigned Word Integer</td>
<td>[0, 65535]</td>
<td>DevSNB+</td>
</tr>
<tr>
<td>W</td>
<td>16</td>
<td>Signed Word Integer</td>
<td>[-32768, 32767]</td>
<td>DevSNB+</td>
</tr>
<tr>
<td>UD</td>
<td>32</td>
<td>Unsigned Doubleword</td>
<td>[0, (2^{32} - 1)]</td>
<td>DevSNB+</td>
</tr>
</tbody>
</table>
**Notation** | **Size in Bits** | **Name** | **Range** | **Generation**
---|---|---|---|---
D | 32 | Signed Doubleword Integer | \([-2^{31}, 2^{31} - 1]\) | DevSNB+
UV | 32 | Packed Unsigned Half-Byte Integer Vector | \([0, 15]\) in each of eight 4-bit immediate vector elements. | DevSNB+
V | 32 | Packed Signed Half-Byte Integer Vector | \([-8, 7]\) in each of eight 4-bit immediate vector elements. | DevSNB+

**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 .

**Floating-Point Numeric Data Types**

The Execution Unit supports the following floating-point data types. 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 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.

In IEEE mode, F calculations flush denormalized values to zero and gradual underflow is not supported.

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.

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

```
3 3 0 2 2 0
S biased exponent fraction
```

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

```
6 6 2 5 5 0
S biased exponent fraction
```
VF: Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number

<table>
<thead>
<tr>
<th>Notation</th>
<th>Size in Bits</th>
<th>Name</th>
<th>Range</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>32</td>
<td>Float</td>
<td>Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand: $[-(2^{31})^{127}...-2^{-149}, 0.0, 2^{-149}... (2^{31})^{127}]$</td>
<td>DevSNB+</td>
</tr>
<tr>
<td>DF</td>
<td>64</td>
<td>Double Float</td>
<td>Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: $[-(2^{52})^{1023}...-2^{1074}, 0.0, 2^{1074}... (2^{52})^{1023}]$</td>
<td>DevIVB+</td>
</tr>
<tr>
<td>VF</td>
<td>32</td>
<td>Packed Restricted Float Vector</td>
<td>Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit, 3 bits for the biased exponent (bias of 3), and 4 bits for the significand: $[-31...-0.125, 0, 0.125... 31]$</td>
<td>DevSNB+</td>
</tr>
</tbody>
</table>

Packed Signed Half-Byte Integer Vector

A packed signed halfbyte integer vector consists of 8 signed halfbyte integers contained in a doubleword. Each signed halfbyte integer element has a range from -8 to 7 with the sign on bit 3. This numeric data type is only used by an immediate source operand of doubleword in a GEN instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN hardware converts the vector into an 8-element signed word vector by sign extension. This is illustrated in *Numeric Data Types*.

The short hand format notation for a packed signed half-byte vector is **V**.

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

A packed unsigned halfbyte integer vector consists of 8 unsigned halfbyte integers contained in a doubleword. Each unsigned halfbyte integer element has a range from 0 to 15. This numeric data type is only used by an immediate source operand of doubleword in a GEN instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN hardware converts the vector into an 8-element signed word vector.
Packed Restricted Float Vector

A packed restricted float vector consists of 4 8-bit restricted floats contained in a doubleword. Each restricted float has the sign at bit 7, a 3-bit coded exponent in bits 4 to 6, a 4-bit fraction in bits 0 to 3, and an implied integer 1. The exponent is in excess-3 format – having a bias of 3. Restricted float provides zero, positive/negative normalized numbers with a small range (3-bit exponent) and small precision (4-bit fraction). This numeric data type is only used by an immediate source operand of doubleword in a GEN instruction. It cannot be used for the destination operand, or a non-immediate source operand.

The following figure shows how to convert an 8-bit restricted float into a single precision float. Converting a 3-bit exponent with a bias of 3 to an 8-bit exponent with a bias of 127 is by adding 4, or equivalently copying bit 2 to bit 7 and putting the inverted bit 2 to bits 6:2. A special logic is also needed to take care of positive/negative zeros.

Conversion from a Restricted 8-bit Float to a Single-Precision Float
The following table shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinites, NaN, and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range – the magnitude of the smallest non-zero number is 0.125.

### Examples of Restricted 8-bit Float Numbers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Normalized</td>
<td>0x70-0x7F</td>
<td>0</td>
<td>111</td>
<td>0000 ... 1111</td>
<td>1000 0011</td>
<td>16 ... 31</td>
</tr>
<tr>
<td>Float</td>
<td>0x60-0x6F</td>
<td>0</td>
<td>110</td>
<td>0000 ... 1111</td>
<td>1000 0010</td>
<td>8 ... 15.5</td>
</tr>
<tr>
<td></td>
<td>0x50-0x5F</td>
<td>0</td>
<td>101</td>
<td>0000 ... 1111</td>
<td>1000 0001</td>
<td>4 ... 7.75</td>
</tr>
<tr>
<td></td>
<td>0x40-0x4F</td>
<td>0</td>
<td>100</td>
<td>0000 ... 1111</td>
<td>1000 0000</td>
<td>2 ... 3.875</td>
</tr>
<tr>
<td></td>
<td>0x30-0x3F</td>
<td>0</td>
<td>011</td>
<td>0000 ... 1111</td>
<td>0111 1111</td>
<td>1 ... 1.9375</td>
</tr>
<tr>
<td></td>
<td>0x20-0x2F</td>
<td>0</td>
<td>010</td>
<td>0000 ... 1111</td>
<td>0111 1110</td>
<td>0.5 ... 0.96875</td>
</tr>
<tr>
<td></td>
<td>0x10-0x1F</td>
<td>0</td>
<td>001</td>
<td>0000 ... 1111</td>
<td>0111 1101</td>
<td>0.25 ... 0.484375</td>
</tr>
<tr>
<td></td>
<td>0x01-0x0F</td>
<td>0</td>
<td>000</td>
<td>0001 ... 1111</td>
<td>0111 1100</td>
<td>0.125 ... 0.2421875</td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td>0</td>
<td>000</td>
<td>0000</td>
<td>0000 0000</td>
<td>0 (+zero)</td>
</tr>
<tr>
<td>Negative Normalized</td>
<td>0xF0-0xFF</td>
<td>1</td>
<td>111</td>
<td>0000 ... 1111</td>
<td>1000 0011</td>
<td>-16 ... -31</td>
</tr>
<tr>
<td>Float</td>
<td>0xE0-0xEF</td>
<td>1</td>
<td>110</td>
<td>0000 ... 1111</td>
<td>1000 0010</td>
<td>-8 ... -15.5</td>
</tr>
<tr>
<td></td>
<td>0xD0-</td>
<td>1</td>
<td>101</td>
<td>0000 ... 1111</td>
<td>1000 0001</td>
<td>-4 ... -7.75</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>----------</td>
<td>----------------</td>
<td>----------------</td>
<td>-------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>0xDF</td>
<td>0x0C0-0xCF</td>
<td>1</td>
<td>100</td>
<td>0000 ... 1111</td>
<td>1000 0000</td>
<td>-2 ... -3.875</td>
</tr>
<tr>
<td></td>
<td>0xB0-0xBF</td>
<td>1</td>
<td>011</td>
<td>0000 ... 1111</td>
<td>0111 1111</td>
<td>-1 ... -1.9375</td>
</tr>
<tr>
<td></td>
<td>0xA0-0xAF</td>
<td>1</td>
<td>010</td>
<td>0000 ... 1111</td>
<td>0111 1110</td>
<td>-0.5 ... -0.96875</td>
</tr>
<tr>
<td></td>
<td>0x90-0x9F</td>
<td>1</td>
<td>001</td>
<td>0000 ... 1111</td>
<td>0111 1101</td>
<td>-0.25 ... -0.484375</td>
</tr>
<tr>
<td></td>
<td>0x81-0x8F</td>
<td>1</td>
<td>000</td>
<td>0001 ... 1111</td>
<td>0111 1100</td>
<td>-0.125 ... -0.2421875</td>
</tr>
<tr>
<td></td>
<td>0x80</td>
<td>1</td>
<td>000</td>
<td>0000</td>
<td>0000 0000</td>
<td>-0 (-zero)</td>
</tr>
</tbody>
</table>

The following figure shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The shorthand format notation for a packed signed half-byte vector is VF.

**Floating Point Modes**

GEN architecture supports two floating point operation modes, namely IEEE floating point mode (IEEE mode) and alternative floating point mode (ALT mode). Both modes follow mostly the requirements in IEEE-754 but with different deviations. The deviations will be described in details in later sections. The primary difference between these modes is on the handling of Infs, NaNs and denorms. The IEEE floating point mode may be used to support newer versions of 3D graphics API Shaders and the alternative floating point mode may be used to support early Shader versions. Taking DirectX 3D graphics API Shaders for example, shader models before version 3.0 may use the alternative floating point mode, while version 3.0 and following shader models may use the IEEE floating point mode.

These two modes are supported by all units that perform floating point computations, including GEN execution units, GEN shared functions like Extended Math, the Sampler and the Render Cache color calculator, and fixed functions like VF, Clipper, SF and WIZ. Host software sets floating point mode through the fixed function state descriptors for 3D pipeline and the interface descriptor for media pipeline. Therefore different modes may be associated with different threads running concurrently.
Floating point mode control for EU and shared functions are based on the floating point mode field (bit 0) of cr0 register.

**IEEE Floating Point Mode**

**Partial Listing of Honored IEEE-754 Rules**

Here is a summary of expected 32-bit floating point behaviors in GEN architecture. Refer to IEEE-754 for topics not mentioned.

- \( \text{INF} - \text{INF} = \text{NaN} \)
- \( 0 \times (+/-)\text{INF} = \text{NaN} \)
- \( 1 / (+\text{INF}) = +0 \) and \( 1 / (-\text{INF}) = -0 \)
  - \( (+/-)\text{INF} / (+/-)\text{INF} = \text{NaN} \) as \( A/B = A \times (1/B) \)
- \( \text{INV} (+0) = \text{RSQ} (+0) = +\text{INF}, \text{INV} (-0) = \text{RSQ} (-0) = -\text{INF}, \) and \( \text{SQRT} (-0) = -0 \)
- \( \text{RSQ} (-\text{finite}) = \text{SQRT} (-\text{finite}) = \text{NaN} \)
- \( \text{LOG} (+0) = \text{LOG} (-0) = -\text{INF}, \text{LOG} (-\text{finite}) = \text{LOG} (-\text{INF}) = \text{NaN} \)
- \( \text{NaN} \) (any OP) any-value = \( \text{NaN} \) with one exception for min/max mentioned below. Resulting \( \text{NaN} \) may have different bit pattern than the source \( \text{NaN} \).
- Normal comparison with conditional modifier of EQ, GT, GE, LT, LE, when either or both operands is \( \text{NaN} \), returns FALSE. Normal comparison of NE, when either or both operands is \( \text{NaN} \), 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 \( \text{NaN} \), returns TRUE, regardless of the first source operand, and when the second source operand is not \( \text{NaN} \), but first one is, returns FALSE. `Cmpn` of NE, when the second source operand is \( \text{NaN} \), returns FALSE, regardless of the first source operand, and when the second source operand is not \( \text{NaN} \), but first one is, returns TRUE.
  - **Note:** Special comparison is used to support the proposed IEEE-754R rule on `min` or `max` operations. For which, if only one operand is \( \text{NaN} \), `min` and `max` operations return the other operand as the result.
- Both normal and special comparisons of any non-\( \text{NaN} \) value against +/- \( \text{INF} \) return exact result according to the conditional modifier. This is because that infinities are exact representation in the sense that \( +\text{INF} = +\text{INF} \) and \( -\text{INF} = -\text{INF} \).
  - \( \text{NaN} \) is unordered in the sense that \( \text{NaN} != \text{NaN} \).
- 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.

The rounding mode is specified by the Rounding Mode field in the Control Register.

The EU can report floating point overflow and NaN into conditional flags. However, there is no support for floating point exceptions, status bits, or traps.

Denoms are handled as follows:

- Single precision (F, Float) denoms are flushed to sign-preserved zero on input and output of any floating-point mathematical operation.
- Double precision (DF, Double Float) denoms are kept or flushed in mathematical operations based on the Double Precision Denorm Mode in the Control Register.
- Denoms are not flushed for format conversions, irrespective of any denorm mode.
- Denoms are not flushed for raw mov operations. For information about raw mov operations, refer to the Description in Instruction Move EUISA.
- Input denoms are not flushed for half precision to single precision floating-point conversion.

Other information regarding floating-point behaviors:

- NaN input to an operation always produces NaN on output, however the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw mov instruction which does not alter data at all.)
- x*1.0f must always result in x (except denorm flushed and possible bit pattern change for NaN).
- x +/- 0.0f must always result in x (except denorm flushed and possible bit pattern change for NaN). But -0 + 0 = +0.
- Fused operations (such as mac, dp4, dp3, etc.) may produce intermediate results out of 32-bit float range, but whose final results would be within 32-bit float range if intermediate results were kept at greater precision. In this case, implementations are permitted to produce either the correct result, or else ±inf. Thus, compatibility between a fused operation, such as mac, with the unfused equivalent, mul followed by add in this case, is not guaranteed.
- As the accumulator registers have more precision than 32-bit float, any instruction with accumulator as a source/destination operand may produce a different result than that using GRF registers.
• API Shader divide operations are implemented as \( x \times (1.0f/y) \). With the two-step method, \( x \times (1.0f/y) \), 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.

Comparison of Floating Point Numbers

The following tables detail the rules for floating point comparison. In the tables, \( +/-\text{Fin} \) stands for a positive or negative finite precision floating point number. Result is either a true (T) or false (F). Each row corresponds to a fixed src0 and each column corresponds to a fixed src1. When comparing two positive finite numbers (or two negative finite numbers), the result can be T or F depending on the values. Therefore, the corresponding fields in the following tables are marked as T/F. When comparing two double float numbers, the result can be T or F depending on the values and the denorm mode (enabled/disabled). The corresponding fields in the following tables are marked T/F*.

### Results of Greater-Than Comparison – CMP.

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td></td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-denorm</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-0</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+0</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+denorm</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+Fin</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>F</td>
</tr>
<tr>
<td>+inf</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

### Results of Less-Than Comparison – CMP.L

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td></td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td></td>
<td>F</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>-denorm</td>
<td></td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>-0</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>+0</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
### Results of Equal-To Comparison – CMP.E

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+denorm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T/F*</td>
<td></td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>+Fin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>F</td>
<td>T/F*</td>
<td>T</td>
</tr>
<tr>
<td>+inf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

### Results of Not-Equal-To Comparison – CMP.NE

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-inf</td>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td></td>
<td>F</td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-denorm</td>
<td></td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-0</td>
<td></td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+0</td>
<td></td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+denorm</td>
<td></td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+Fin</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T/F*</td>
<td>F</td>
</tr>
<tr>
<td>+inf</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

### Results of Less-Than Or Equal-To Comparison – CMP.LE

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-inf</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td></td>
<td>F</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

990
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.

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'.

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

---

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td>T</td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-denorm</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-0</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+0</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+denorm</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+Fin</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+inf</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NaN</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Results of Greater-Than or Equal-To Comparison – CMP.GE

<table>
<thead>
<tr>
<th>src0</th>
<th>src1</th>
<th>-inf</th>
<th>-Fin</th>
<th>-denorm</th>
<th>-0</th>
<th>+0</th>
<th>+denorm</th>
<th>+Fin</th>
<th>+inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-Fin</td>
<td>T</td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-denorm</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>-0</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+0</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+denorm</td>
<td>T</td>
<td>T</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>T/F*</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+Fin</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>+inf</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NaN</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

---
floating point number, i.e., $+\text{fmax}$. 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

<table>
<thead>
<tr>
<th>IEEE-754 Deviations</th>
<th>VF</th>
<th>Clipper</th>
<th>SF</th>
<th>WIZ</th>
<th>EU</th>
<th>EM</th>
<th>Sampler</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any +/- INF result flushed to +/- fmax</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Log, rsq, sqrt take abs() of sources</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Y</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*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

<table>
<thead>
<tr>
<th>Suggested IEEE-754 Deviations</th>
<th>Reason for Dismiss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov forces (+/-)INF to (+/-)fmax</td>
<td>(+/-)INF is never present as input</td>
</tr>
<tr>
<td>(+/-)INF – (+/-)INF = +/- fmax instead of NaN</td>
<td>(+/-)INF is never present as input</td>
</tr>
<tr>
<td>Denorm must be flushed to zero in all cases (including trivial mov and point sampling)</td>
<td>Denorm is never present as input</td>
</tr>
<tr>
<td>Anything<em>0=0 (including NaN</em>0=0 and INF*0=0)</td>
<td>NaN and INF are never present as input</td>
</tr>
<tr>
<td>Except propagated NaN, NaN is never generated</td>
<td>NaN is never present as input and GEN never generates NaN based on rules in the previous table</td>
</tr>
<tr>
<td>An input NaN gets propagated excepting (a)-(d)</td>
<td>NaN is never present as input</td>
</tr>
<tr>
<td>(a) Rcp (and rsq) of 0 yields fmax</td>
<td>N/A, as it is already covered by the general rule <em>Any +/- INF result flushed to +/- fmax</em></td>
</tr>
<tr>
<td>(b) Sampler honors 0/0 = 0 as if (1/0)*0</td>
<td>There is no divide in Sampler</td>
</tr>
<tr>
<td>(c) Rcp (and rsq) of INF yields +/- 0</td>
<td>(+/-)INF is never present as input</td>
</tr>
<tr>
<td>Suggested IEEE-754 Deviations</td>
<td>Reason for Dismiss</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0</td>
<td>There is no divide in Sampler</td>
</tr>
</tbody>
</table>
Type Conversion

Float to Integer

Converting from float to integer is based on rounding toward zero (RTZ is for DX, IEEE expects all four rounding modes). If the floating point value is +0, -0, +Denorm, -Denorm, +NaN –r -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) representable integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) representable integer value, the conversion result takes the largest (or the smallest) representable integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the representable range of the output integer type (UD, D, UW, W, UB and B).

<table>
<thead>
<tr>
<th>Input Format</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>UD</td>
</tr>
<tr>
<td>+/- Zero</td>
<td>00000000</td>
</tr>
<tr>
<td>+/- Denorm</td>
<td>00000000</td>
</tr>
<tr>
<td>NAN</td>
<td>00000000</td>
</tr>
<tr>
<td>-NAN</td>
<td>00000000</td>
</tr>
<tr>
<td>INF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>-INF</td>
<td>00000000</td>
</tr>
<tr>
<td>+2^{32} (*)</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>-2^{32-1} (*)</td>
<td>00000000</td>
</tr>
</tbody>
</table>

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 representable number becomes a negative number. With saturation, a larger than representable number is saturated to the largest positive representable 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**

<table>
<thead>
<tr>
<th>Double Precision Float</th>
<th>Single Precision Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>-inf</td>
</tr>
<tr>
<td>-finite</td>
<td>-finite/-denorm/-0</td>
</tr>
<tr>
<td>-denorm</td>
<td>-0</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+denorm</td>
<td>+0</td>
</tr>
<tr>
<td>+finite</td>
<td>+finite/+denorm/+0</td>
</tr>
<tr>
<td>+inf</td>
<td>+inf</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Single Precision Float</th>
<th>Double Precision Float</th>
</tr>
</thead>
</table>

995
<table>
<thead>
<tr>
<th>Single Precision Float</th>
<th>Double Precision Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>-inf</td>
</tr>
<tr>
<td>-finite</td>
<td>-finite</td>
</tr>
<tr>
<td>-denorm</td>
<td>-finite</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+denorm</td>
<td>+finite</td>
</tr>
<tr>
<td>+finite</td>
<td>+finite</td>
</tr>
<tr>
<td>+inf</td>
<td>+inf</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Trigger / Source</th>
<th>Sync/Async Recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Exception</td>
<td>Thread code</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>• A bit in the instruction word</td>
<td>Synchronous</td>
</tr>
<tr>
<td></td>
<td>• Breakpoint IP match</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Breakpoint Opcode match</td>
<td></td>
</tr>
<tr>
<td>Illegal Opcode</td>
<td>Hardware</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Halt</td>
<td>MMIO register write</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Context Save/Restore</td>
<td>Preemption Interrupt</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

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.

When the System Routine is invoked, the application thread’s current IP is first saved into the AIP field of cr0.2. The SIP address is then loaded into the thread’s IP register and execution continues within the System Routine. Thus each invocation of the System Routine has a common entry point. Returning from the System Routine loads IP from AIP, continuing thread execution.

**System Routine Register Space**

The System Routine uses the same GRF space as the thread that invokes it. As such all of the calling thread’s registers and their contents are visible to the System Routine. Further, the System Routine must only use r0..r15 of the GRF, as a minimal thread may have requested and been allocated this few. If the System Routine requires more registers than this, the driver should establish a higher minimum allocation for all threads.

The System Routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

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 provided increased performance when a kernel can guarantee no dependency issues between instructions, but may cause issues with exception handling in some circumstances as discussed here.

Typically NoDDClr is used in an instruction series to enable a sequence of writes to sub-fields of a GRF register without paying a dependency penalty on each instruction. In this case, NoDDClr and NoDDChk are used across an instruction sequence to allow the first instruction to set the destination dependency, interior instructions to write to the GRF register without dependency checks, and the last instruction to clear the dependency. (This sequence is referred to as a NoDDClr code block going forward). By only allowing the last instruction to clear the dependency, program execution is prevented from going beyond a certain point until all writes of that sequence are known to retire.

The problem arises if an exception is raised within a NoDDClr code block. In this case, there exists the potential for the System Routine to hang while attempting to save/restore a register used as a destination register by the NoDDClr code block, as the outstanding dependency on that register will not clear until the final instruction of the NoDDClr block is executed, sometime after the System Routine returns. Should the System Routine attempt to use that register, it hangs waiting on a dependency to be cleared by an instruction not yet issued.

**Note:** This is a known condition and will in some cases not allow the full GRF contents to be externally visible in System Routine scratch space during a break or halt exception.

To avoid this condition, guidelines are provided below for consideration. (Note that these are general guidelines, some of which can be alleviated through careful coding and register usage conventions and restrictions.)

- NoDDClr code blocks should only be used where absolutely necessary.
- Instructions that may generate exceptions should not be placed within NoDDClr blocks. This includes most conditional branch instructions (if, do, while, ...) .
- If possible, use NoDDClr on registers high in the thread’s register allocation (e.g. r120), thus even if a System Routine hang occurs, as much of the GRF is visible as possible. (Note that this would also...
require the System Routine to update the progress of the GRF dump, perhaps with each GRF block written, or to initialize the System Routine’s scratch space to a known value, to be able to distinguish valid/locations from unwritten locations).

Also a driver implementation may consider a disable-NoDDClr option in which jitted code does not use the NoDDClr capability. In this case, there is no change to the code that is jitted other than removal of the NoDDClr instruction option. The code executes as normal, but with a higher number of thread switches in what would have been a NoDDClr code block.
Exception Descriptions

This section describes conditions that can cause exceptions and transfer control to the System Routine.

Illegal Opcode

The GEN ISA defines a single illegal opcode. The byte value of the illegal opcode is 0x00 due to it being a likely byte value encountered by a wayward instruction pointer value. The illegal instruction signals an exception if exception handling is enabled and invokes the system interrupt routine. If exception handling is NOT enabled, the illegal opcode is executed resulting in undetermined behavior including a system hang. Hardware decodes all legal opcodes supported. Any byte value that is not in the legal opcode list is decoded as an illegal opcode to trigger exception.

Undefined Opcodes

All undefined opcodes in the 8-bit opcode space (which includes instruction bit 7, reserved for future opcode expansion) are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing the opcode value within the pipeline to the defined illegal opcode. The offending instruction, should it eventually be issued down the execution unit's pipeline, generates an Illegal Opcode exception as described in the section Illegal Opcode. The memory location of the offending opcode keeps its original value. That location can be queried to determine the opcode value.

Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered using the Software Exception Set and Clear bit of cr0.1. Sub-function determination and parameter passing into and out of the exception handler is left to convention between the system-thread and application-thread. The thread's IP is incremented before saving AIP and entering the System Routine, causing execution to resume at the next application-thread instruction after returning from the System Routine.

Context Save and Restore

The System Routine is also used to save and restore the context of the Execution Unit. This feature is enabled in GPGPU workloads only.

When the execution engine receives a preemption or an interrupt, the application thread invokes the System Routine. The System Routine is invoked only when all in-flight registers have retired. The system routine is used to save all the state of the EU to memory. When the sequence is complete, the master exception control bit is cleared. This action stops all execution for the given thread and invalidates the thread. This means a new thread from a different context may be loaded. When the master exception control bit is cleared, software must ensure that all outstanding messages from the EU are dispatched out of the execution message pipeline. This is achieved by creating a dependency on the last send that is saving EU state. A dummy instruction before clearing the master exception control bit ensures that this is achieved.
The System Routine is also invoked on a context restore request. In this case a dummy thread is loaded into the EU which starts with the System Routine. This routine now restores the state of the EU. The restore sequence used in such a case should be consistent with the save sequence to ensure that state is restored correctly. After completing the restore sequence, the System Routine must clear the master exception control bit in the Control Register. This enables hardware to switch to the application thread which continues execution.
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.

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 (\texttt{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.
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. See SIMD Instructions and SIMD Width for parameters for difference device hardware.

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. [HSW]: There are 8 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 Machine 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).

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.
• 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 two diagrams are for native instructions with one or two source operands.
## GEN Instruction Format – 1-src and 2-src

<table>
<thead>
<tr>
<th>INX Bits</th>
<th>High Byte</th>
<th>Low Byte</th>
<th>Intr bits</th>
<th>Add/Mode</th>
<th>Access/Indirect</th>
<th>SEND</th>
<th>Branch</th>
<th>Imm Src</th>
<th>IMM</th>
<th>DIAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>128</td>
<td>127</td>
<td>1</td>
<td>127</td>
<td>127</td>
<td>EOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>126</td>
<td>125</td>
<td>2</td>
<td>125</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>124</td>
<td>123</td>
<td>3</td>
<td>123</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td>117</td>
<td>4</td>
<td>117</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>116</td>
<td>115</td>
<td>5</td>
<td>115</td>
<td>115</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>113</td>
<td>112</td>
<td>6</td>
<td>112</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>110</td>
<td>7</td>
<td>110</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>109</td>
<td>108</td>
<td>8</td>
<td>108</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>106</td>
<td>105</td>
<td>9</td>
<td>105</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>103</td>
<td>102</td>
<td>10</td>
<td>102</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>99</td>
<td>98</td>
<td>11</td>
<td>98</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>96</td>
<td>95</td>
<td>12</td>
<td>95</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>93</td>
<td>92</td>
<td>13</td>
<td>92</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>90</td>
<td>89</td>
<td>14</td>
<td>89</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>86</td>
<td>85</td>
<td>15</td>
<td>85</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>82</td>
<td>81</td>
<td>16</td>
<td>81</td>
<td>81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>79</td>
<td>78</td>
<td>17</td>
<td>78</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>76</td>
<td>75</td>
<td>18</td>
<td>75</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>73</td>
<td>72</td>
<td>19</td>
<td>72</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>69</td>
<td>68</td>
<td>20</td>
<td>68</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>66</td>
<td>65</td>
<td>21</td>
<td>65</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>63</td>
<td>62</td>
<td>22</td>
<td>62</td>
<td>62</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>60</td>
<td>59</td>
<td>23</td>
<td>59</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>57</td>
<td>56</td>
<td>24</td>
<td>56</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>54</td>
<td>53</td>
<td>25</td>
<td>53</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>51</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>48</td>
<td>47</td>
<td>27</td>
<td>47</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>45</td>
<td>44</td>
<td>28</td>
<td>44</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>42</td>
<td>41</td>
<td>29</td>
<td>41</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>39</td>
<td>38</td>
<td>30</td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>36</td>
<td>35</td>
<td>31</td>
<td>35</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>30</td>
<td>29</td>
<td>33</td>
<td>29</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>28</td>
<td>27</td>
<td>34</td>
<td>27</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>25</td>
<td>24</td>
<td>35</td>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>23</td>
<td>22</td>
<td>36</td>
<td>22</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>20</td>
<td>19</td>
<td>37</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>17</td>
<td>16</td>
<td>38</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>14</td>
<td>13</td>
<td>39</td>
<td>13</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>11</td>
<td>10</td>
<td>40</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>9</td>
<td>8</td>
<td>41</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>6</td>
<td>5</td>
<td>42</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td>2</td>
<td>43</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Reserved for Opcode | 0 |

**Legend:**
- **EOT**: End Of Table
- **MATH**: Math
- **Reg**: Register
- **Imm**: Immediate
- **DIAM**: DIAM
- **END**: End of Table
The next two diagrams are for instructions with three source operands.

**GEN Instruction Format – 3-src**

<table>
<thead>
<tr>
<th>DW #</th>
<th>Instr Bits</th>
<th>High Bit</th>
<th>Low Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>127</td>
<td>128</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>8</td>
<td>125</td>
<td>118</td>
<td>8</td>
<td>Src2 Regnum</td>
</tr>
<tr>
<td>3</td>
<td>117</td>
<td>115</td>
<td>3</td>
<td>Src2 Subregnum</td>
</tr>
<tr>
<td>8</td>
<td>114</td>
<td>107</td>
<td>8</td>
<td>Src2 Swizzle</td>
</tr>
<tr>
<td>1</td>
<td>106</td>
<td>106</td>
<td>1</td>
<td>Src2 RepCtrl</td>
</tr>
<tr>
<td>1</td>
<td>105</td>
<td>105</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>8</td>
<td>104</td>
<td>97</td>
<td>8</td>
<td>Src1 Regnum</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
<td>94</td>
<td>3</td>
<td>Src1 Subregnum</td>
</tr>
<tr>
<td>8</td>
<td>93</td>
<td>86</td>
<td>8</td>
<td>Src1 Swizzle</td>
</tr>
<tr>
<td>1</td>
<td>85</td>
<td>85</td>
<td>1</td>
<td>Src1 RepCtrl</td>
</tr>
<tr>
<td>1</td>
<td>84</td>
<td>84</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>83</td>
<td>76</td>
<td>9</td>
<td>Src0 Regnum</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>73</td>
<td>3</td>
<td>Src0 Subregnum</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>65</td>
<td>8</td>
<td>Src0 Swizzle</td>
</tr>
<tr>
<td>1</td>
<td>04</td>
<td>04</td>
<td>1</td>
<td>Src0 RepCtrl</td>
</tr>
<tr>
<td>2,3</td>
<td>63</td>
<td>56</td>
<td>8</td>
<td>DstRegnum</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
<td>53</td>
<td>3</td>
<td>Dst Subregnum</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>48</td>
<td>4</td>
<td>Dst chan enable</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>47</td>
<td>1</td>
<td>NbCtrl</td>
</tr>
<tr>
<td>1</td>
<td>46</td>
<td>46</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>44</td>
<td>2</td>
<td>Dst Type</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td>42</td>
<td>2</td>
<td>Src Type</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>40</td>
<td>2</td>
<td>Src2 Modifier</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>38</td>
<td>2</td>
<td>Src1 Modifier</td>
</tr>
<tr>
<td>2</td>
<td>37</td>
<td>36</td>
<td>2</td>
<td>Src0 Modifier</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>35</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>34</td>
<td>1</td>
<td>FlagRegNum</td>
</tr>
<tr>
<td>1</td>
<td>33</td>
<td>33</td>
<td>1</td>
<td>Flag Sub Req Num</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>32</td>
<td>1</td>
<td>reserved</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>31</td>
<td>1</td>
<td>Saturate</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>30</td>
<td>1</td>
<td>Debug Ctrl</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>29</td>
<td>1</td>
<td>CsH Ctrl</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>28</td>
<td>1</td>
<td>AccW Ctrl</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>24</td>
<td>4</td>
<td>CondModifier</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>21</td>
<td>3</td>
<td>Exc Size</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
<td>1</td>
<td>Pref lin</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>16</td>
<td>4</td>
<td>Pedi Ctrl</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>14</td>
<td>2</td>
<td>Thread Ctrl</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>12</td>
<td>2</td>
<td>Qri Ctrl</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>10</td>
<td>2</td>
<td>Dep Ctrl</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>WE Ctrl</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>Access Mode</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>(reserved for Opcode)</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>Opcode</td>
</tr>
</tbody>
</table>

**EU Instruction Formats**

This section describes the Execution Unit instruction formats.

This section covers the layout of instruction fields, not changes in allowed field encodings from generation to generation.
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 exceptional instructions.

**Execution Unit Instruction Format for 1 or 2 Source Operands**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>AddrMode and AccessMode Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AddrMode = Direct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Align16</td>
</tr>
<tr>
<td>127:121</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>120:117</td>
<td>Src1.VertStride</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>Varies based on AccessMode</td>
<td>Reserved</td>
</tr>
<tr>
<td>113:112</td>
<td>Src1.Width</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Src1.AddrMode</td>
<td></td>
</tr>
<tr>
<td>110:109</td>
<td>Src1.SrcMod</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99:96</td>
<td>Src1.ChanSel[3:0]</td>
<td></td>
</tr>
<tr>
<td>95:91</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>FlagRegNum</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>FlagSubRegNum</td>
<td></td>
</tr>
<tr>
<td>88:85</td>
<td>Src0.VertStride</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>Varies based on AccessMode</td>
<td>Reserved</td>
</tr>
<tr>
<td>83:82</td>
<td>Src0.ChanSel[7:4]</td>
<td></td>
</tr>
<tr>
<td>81:80</td>
<td>Src0.HorzStride</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>Src0.AddrMode</td>
<td></td>
</tr>
<tr>
<td>78:77</td>
<td>Src0.SrcMod</td>
<td></td>
</tr>
<tr>
<td>76:74</td>
<td>Varies based on AddrMode and AccessMode</td>
<td>Src0.RegNum</td>
</tr>
<tr>
<td>68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67:64</td>
<td>Src0.ChanSel[3:0]</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Dst.AddrMode</td>
<td></td>
</tr>
</tbody>
</table>
### AddrMode and AccessMode Variations

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>AddrMode = Direct</th>
<th>AddrMode = Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td></td>
<td></td>
<td>Dst.AddrImm[9:0]</td>
</tr>
<tr>
<td>47</td>
<td>HSW: NibCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:44</td>
<td>Src1.SrcType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43:42</td>
<td>Src1.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41:39</td>
<td>Src0.SrcType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38:37</td>
<td>Src0.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36:34</td>
<td>Dst.DstType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33:32</td>
<td>Dst.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>CmptCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>AccWrCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>CondModifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:21</td>
<td>ExecSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>PredInv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:16</td>
<td>PredCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>ThreadCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td>QtrCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:10</td>
<td>DepCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MaskCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AccessMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved (for future Opcode expansion)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:0</td>
<td>Opcode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 3-source operand instructions are:

- **bfe** - Bit Field Extract
- **bfi2** - Bit Field Insert 2
- **lrp** - Linear Interpolation
- **mad** - Multiply Add
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.

**Execution Unit Instruction Format for 3 Source Operands**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>127:126</td>
<td>Reserved</td>
</tr>
<tr>
<td>125:118</td>
<td>Src2.RegNum</td>
</tr>
<tr>
<td>117:115</td>
<td>Src2.SubRegNum</td>
</tr>
<tr>
<td>114:107</td>
<td>Src2.ChanSel</td>
</tr>
<tr>
<td>106</td>
<td>Src2.RepCtrl</td>
</tr>
<tr>
<td>105</td>
<td>Reserved</td>
</tr>
<tr>
<td>104:97</td>
<td>Src1.RegNum</td>
</tr>
<tr>
<td>96</td>
<td>Src1.SubRegNum[2]</td>
</tr>
<tr>
<td>95:94</td>
<td>Src1.SubRegNum[1:0]</td>
</tr>
<tr>
<td>93:86</td>
<td>Src1.ChanSel</td>
</tr>
<tr>
<td>85</td>
<td>Src1.RepCtrl</td>
</tr>
<tr>
<td>84</td>
<td>Reserved</td>
</tr>
<tr>
<td>83:76</td>
<td>Src0.RegNum</td>
</tr>
<tr>
<td>75:73</td>
<td>Src0.SubRegNum</td>
</tr>
<tr>
<td>72:65</td>
<td>Src0.ChanSel</td>
</tr>
<tr>
<td>64</td>
<td>Src0.RepCtrl</td>
</tr>
<tr>
<td>63:56</td>
<td>Dst.RegNum</td>
</tr>
<tr>
<td>55:53</td>
<td>Dst.SubRegNum</td>
</tr>
<tr>
<td>52:49</td>
<td>Dst.ChanEnable</td>
</tr>
<tr>
<td>48</td>
<td>Reserved</td>
</tr>
<tr>
<td>47</td>
<td>NibCtrl</td>
</tr>
<tr>
<td>46</td>
<td>Reserved</td>
</tr>
<tr>
<td>45:44</td>
<td>DstType</td>
</tr>
<tr>
<td>43:42</td>
<td>SrcType</td>
</tr>
<tr>
<td>41:40</td>
<td>Src2.Modifier</td>
</tr>
<tr>
<td>39:38</td>
<td>Src1.Modifier</td>
</tr>
<tr>
<td>37:36</td>
<td>Src0.Modifier</td>
</tr>
<tr>
<td>35</td>
<td>Reserved</td>
</tr>
<tr>
<td>34</td>
<td>FlagRegNum</td>
</tr>
</tbody>
</table>
### Bits and Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>FlagSubRegNum</td>
</tr>
<tr>
<td>32</td>
<td>HSW: Reserved</td>
</tr>
<tr>
<td>31</td>
<td>Saturate</td>
</tr>
<tr>
<td>29</td>
<td>CmptCtrl</td>
</tr>
<tr>
<td>28</td>
<td>AccWrCtrl</td>
</tr>
<tr>
<td>27:24</td>
<td>CondModifier</td>
</tr>
<tr>
<td>23:21</td>
<td>ExecSize</td>
</tr>
<tr>
<td>20</td>
<td>PredInv</td>
</tr>
<tr>
<td>19:16</td>
<td>PredCtrl</td>
</tr>
<tr>
<td>15:14</td>
<td>ThreadCtrl</td>
</tr>
<tr>
<td>13:12</td>
<td>QtrCtrl</td>
</tr>
<tr>
<td>11:10</td>
<td>DepCtrl</td>
</tr>
<tr>
<td>9</td>
<td>MaskCtrl</td>
</tr>
<tr>
<td>8</td>
<td>AccessMode</td>
</tr>
<tr>
<td>7</td>
<td>Reserved (for future Opcode expansion)</td>
</tr>
<tr>
<td>6:0</td>
<td>Opcode</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Regular 1 or 2 Source Operands Description</th>
<th>Empty white areas mean Same, use the regular description</th>
<th>Branch Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>Reserved</td>
<td>EOT</td>
<td>UIP[15:0] (2-offset branches)</td>
</tr>
<tr>
<td>126:125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124:121</td>
<td></td>
<td>Imm[28:0] / Reg32</td>
<td></td>
</tr>
<tr>
<td>120:117</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116:112</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110:109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108:96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95:91</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>FlagRegNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits</td>
<td><strong>Regular 1 or 2 Source Operands Description</strong></td>
<td><strong>Branch Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------</td>
<td>-------------------------</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>FlagSubRegNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88:85</td>
<td>Src0.VertStride</td>
<td>math</td>
<td></td>
</tr>
<tr>
<td>84:80</td>
<td>Varies based on AccessMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>Src0.AddrMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78:77</td>
<td>Src0.SrcMod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76:64</td>
<td>Varies based on AddrMode and AccessMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Dst.AddrMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:61</td>
<td>Varies based on AccessMode</td>
<td>Any branch instruction: Same as regular</td>
<td></td>
</tr>
<tr>
<td>60:48</td>
<td>Varies based on AddrMode and AccessMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>HSW: NibCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:44</td>
<td>Src1.SrcType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43:42</td>
<td>Src1.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41:39</td>
<td>Src0.SrcType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38:37</td>
<td>Src0.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36:34</td>
<td>Dst.DstType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33:32</td>
<td>Dst.RegFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>CmptCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>AccWrCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>CondModifier</td>
<td>SFID[3:0] FC[3:0]</td>
<td></td>
</tr>
<tr>
<td>23:21</td>
<td>ExecSize</td>
<td>Any branch instruction: MBZ</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>PredInv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:16</td>
<td>PredCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>ThreadCtrl</td>
<td>Same as regular</td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td>QtrCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:10</td>
<td>DepCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MaskCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AccessMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved (for future Opcode expansion)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Common Instruction Fields

As shown in the table below, the meanings (encoding) of certain bit fields in the 128-bit native instruction format varies depending on the values of other bit fields.

Definitions of Common Instruction Fields (below) provides the definition of common fields in the native instruction format. The Width column specifies the width of the field in bits. These common fields are referenced in describing the fields of different doublewords of the instruction. The definition for fields that have unique representations can be found in the sections for the corresponding instruction DWords.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>CondModifier</td>
<td><strong>Conditional Modifier.</strong> This field sets the flag register based on the internal conditional signals output from the execution pipe such as sign, zero, overflow and NaNs, etc. If this field is set to 0000, no flag registers are updated. Flag registers are not updated for instructions with embedded compares. This field applies to all instructions except send, sendc, and math.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0000 = Do not modify the flag register (normal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001 = Zero or Equal (.z or .e)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010 = Not Zero or Not Equal (.nz or .ne)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011 = Greater-than (.g)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100 = Greater-than-or-equal (.ge)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101 = Less-than (.l)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110 = Less-than-or-equal (.le)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 = Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000 = Overflow (signed overflow) (.o)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1001 = Unordered with Computed NaN (.u)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1010 -1111 = Reserved</td>
<td></td>
</tr>
<tr>
<td>AddrMode</td>
<td><strong>Addressing Mode.</strong> This field determines the addressing method of the operand. Normally the destination operand and each source operand each have a distinct addressing mode field. When it is cleared, the register address of the operand is directly provided by bits in the instruction word. It is called a direct register addressing mode. When it is set, the register</td>
<td>1</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Width</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>address of the operand is computed based on the address register value and an address immediate field in the instruction word. This is referred to as a register-indirect register addressing mode. This field applies to the destination operand and the first source operand, src0. Support for src1 is device dependent. See Table XX (Indirect source addressing support available in device hardware) in ISA Execution Environment for details. 0 = Direct. Direct register addressing 1 = Register-Indirect (or in short Indirect). Register-indirect register addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegNum</td>
<td><strong>Register Number.</strong> This field provides the register number for the operand. For GRF register operand, it provides the portion of register address aligning to 256-bit. For an ARF register operand, this field is encoded such that MSBs identify the architecture register type and LSBs provide its register number. This field together with the corresponding SubRegNum field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [12:5] of the byte address, while SubRegNum field provides bits [4:0]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U8, if RegFile = GRF. 0x00 to 0x7F = Register number in the range of [0, 127] 0x80 to 0xFF = Reserved Format = U8 0x00 to 0x0F = Register number in the range of [0, 15] 0x10 to 0xFF = Reserved Format = 8-bit encoding, if RegFile = ARF. This field is used to encode the architecture register as well as providing the register number. See GEN Execution Environment chapter for details.</td>
<td>8</td>
</tr>
<tr>
<td>SubRegNum</td>
<td><strong>Sub-Register Number.</strong> This field provides the sub-register number for the operand. For a GRF register operand, it provides the byte address within a 256-bit register. For an ARF register operand, this field also provides the sub-register number according to the encoding defined for the given architecture register. This field together with the corresponding RegNum field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [4:0] of the byte address, while the RegNum field provides bits [12:5]. This field applies to the destination operand and the source operands. It is ignored (or</td>
<td>5</td>
</tr>
</tbody>
</table>
Field | Description | Width
--- | --- | ---
not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. | | |
**Note:** The recommended instruction syntax uses sub-register 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 sub-register numbers 0 to 7, corresponding to sub-register byte addresses of 0 to 28 in steps of 4, the element size. | |
Format = U5, if $RegFile = GRF$ 0x00 to 0x1F = Sub-Register number in the range of [0, 31] Format = 5-bit encoding, if $RegFile = ARF$. This field is used to encode the architecture register as well as providing the register number. See GEN Execution Environment chapter for details. | |
AddrSubRegNum | **Address Sub-Register Number.** This field provides the sub-register number for the address register. The address register contains 8 sub-registers. The size of each sub-register is one word. The address register contains the register address of the operand, when the operand is in register-indirect addressing mode. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed. An address sub-register used for indirect addressing is often called an **index register**. Format = U3 0x0 to 0x7 = Address Sub-Register number in the range [0, 7] | 3 |
AddrImm | **Address Immediate.** This field provides the immediate value in units of bytes added to the address register to compute the register address (byte-aligned region origin) for the operand. It is a signed integer. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed.  
*Note: that the address immediate field may not be able to cover the whole GRF register range for a thread, as the maximum GRF register space for a thread is 4KB.* Format = S9 Valid range: [-512, 511] | 10 |
SrcMod | **Source Modifier.** This field specifies the numeric modification of a source operand. The value of each data element of a source operand can optionally have its absolute value taken and/or its sign inverted prior to delivery to the execution pipe. The absolute value | 2 |
is prior to negate such that a guaranteed negative value can be produced.
This field only applies to source operand. It does not apply to destination.
This field is not present for an immediate source operand.
00 = No modification (normal)
01 = (abs). Absolute
10 = -. Negate
11 = -(abs). Negate of the absolute (forced negative value)

VertStride

**Vertical Stride.** The field provides the vertical stride of the register region in unit of data elements for an operand.

Encoding of this field provides values of 0 or powers of 2, ranging from 1 to 32 elements. Larger values are not supported due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total).

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 the addresses. Each address sub-register provides the origin for a row of data element. The number of address sub-registers used is determined by the division of ExecSize of the instruction by the Width fields of the operand.

This field only applies to source operand. It does not apply to destination.
This field is not present for an immediate source operand.

For **Align16** access mode, only encodings of 0000, 0010 and 0011 are allowed. Other codes are reserved.

**Note 1:** 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).

**Note 2:** In Align16 access mode, as encoding 0xF is reserved, only single-index indirect addressing is supported.

**Note 3:** If indirect address is supported for src1, encoding 0xF is reserved for src1 - only single-index indirect addressing is supported.

**Note 4:** Encoding 0010 applies for QWord-size operands.

<p>| 0000 = 0 Elements |
| 0001 = 1 Element  |
| 0010 = 2 Elements |
| 0011 = 4 Elements |
| 0100 = 8 Elements |
| 0101 = 16 Elements (applies to byte or word operand only) |</p>
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Width</strong>. This field specifies the number of elements in the horizontal dimension of the region for a source operand. This field cannot exceed the ExecSize field of the instruction. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 000 = 1 Elements 001 = 2 Elements 010 = 4 Elements 011 = 8 Elements 100 = 16 Elements 101-111 = Reserved</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><strong>HorzStride</strong>. This field provides the distance in unit of data elements between two adjacent data elements within a row (horizontal) in the register region for the operand. This field applies to both destination and source operands. This field is not present for an immediate source operand. 00 = 0 Elements 01 = 1 Element 10 = 2 Elements 11 = 4 Elements</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>Imm32</strong>. The 32-bit immediate data field for the operand. It may contain any legal bit pattern for its associated type. Only one 32-bit immediate value may be present in an instruction, therefore binary operations only support src1 as an immediate value. The low order bits are directly used when fewer than 32-bits are needed to describe the desired type; the 32-bits are not coerced into the designated type. For UW and W data types, programmer is required to replicate the lower word to the upper word of this field. This field only applies to the last source operand. Signed and unsigned byte integer data types are not supported for an immediate operand.</td>
<td>32</td>
</tr>
</tbody>
</table>
### Field | Description | Width
--- | --- | ---
See the **Numeric Data Types** section for information about data types and their ranges.

| ChanEn | **Channel Enable.** Four channel enables are defined for controlling which channels will be written into the destination region. These channel mask bits are applied in a modulo-four manner to all *ExecSize* channels. There is 1-bit Channel Enable for each channel within the group of 4. If the bit is cleared, the write for the corresponding channel is disabled. If the bit is set, the write is enabled. Mnemonic for the bit being set for the group of 4 is *x*, *y*, *z*, and *w*, respectively, where *x* corresponds to Channel 0 in the group and *w* corresponds to channel 3 in the group. 
This field only applies to destination operand. 
This field is only present in **Align16** mode. 
0 = Write Disabled 
1 = Write Enabled (normal) | 4 |

| ChanSel | **Channel Select.** This field controls the channel swizzle for a source operand. 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. 
For example with an execution size of 8, *r0.0<4>.zywzf* would assign the channels as follows: Chan0 = Data2, Chan1 = Data1, Chan2 = Data3, Chan3 = Data2; Chan4 = Data6, Chan5 = Data5, Chan6 = Data7, Chan7 = Data6. 
This field only applies to source operand. 
This field is only present in **Align16** mode. It is not present for an immediate source operand. 
The 2-bit Channel Selection field for each channel within the group of 4 is defined as the following.
00 = *x*. Channel 0 is selected for the corresponding execution channel 
01 = *y*. Channel 1 is selected for the corresponding execution channel 
10 = *z*. Channel 2 is selected for the corresponding execution channel 
11 = *w*. Channel 3 is selected for the corresponding execution channel | 8 |

| RepCtrl | **Replicate Control.** This field controls the replication of the starting channel to all channels in the execution size. 
This field applies to all three source operands. 
0 = No replication 
1 = Replicate across all channels | 1 |
Field description:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgDscpt31</td>
<td><strong>Message Description.</strong> This field, containing 31-bit immediate values, provides the description of the message to be sent. This field only applies to the <em>send</em> instruction. It is not present for other instructions. The meaning of the field depends on the type of message as well as the message shared function target. Format: U31</td>
<td>31</td>
</tr>
<tr>
<td>EOT</td>
<td><strong>End of Thread.</strong> This field controls the termination of the thread. For a <em>send</em> instruction, if this field is set, EU will terminate the thread and also set the EOT bit in the message sideband. This field only applies to the <em>send</em> instruction. It is not present for other instructions. 0 = The thread is not terminated 1 = EOT</td>
<td>1</td>
</tr>
</tbody>
</table>

**Instruction Operation Doubleword (DW0)**

Most fields in Instruction Operation Doubleword (DW0) apply to all instructions. Bit field [27:24] is one exception. It is *CondModifier* for most instructions but is *SFID[3:0]* field for the *send* instruction.

The descriptions in the table below are shared between the 1-src/2-src instructions and 3-src instructions.

**Definitions of Fields in Operation Doubleword (DW0)**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td><strong>Saturate.</strong> This field controls the destination saturation.</td>
</tr>
<tr>
<td></td>
<td>When it is set, output data to the destination register are saturated. The saturation operation depends on the destination data type. Saturation is the operation that converts any data that is outside the saturation target range for the data type to the closest represented value with the target range. If destination type is float, saturation target range is [0, 1]. For example, any positive number greater than 1 (including +INF) is saturated to 1 and any negative number (including -INF) is saturated to 0. A NaN is saturated to 0. For integer data types, the maximum range for the given numeric data type is the saturation target range. When it is not set, output data to the destination register are not saturated. For example, a wrapped result (modular) is output to the destination for an overflowed integer data. More details can be found in the Data Types chapter. 0 = No destination modification (normal) 1 = sat. Saturate the output</td>
</tr>
</tbody>
</table>

<p>| Destination Type | Saturation Target Range (inclusive) |</p>
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float (F)</td>
<td>[0.0, 1.0]</td>
</tr>
<tr>
<td>Byte (UB)</td>
<td>[0, 255]</td>
</tr>
<tr>
<td>Signed Byte (B)</td>
<td>[-128, 127]</td>
</tr>
<tr>
<td>Word (UW)</td>
<td>[0, 65535]</td>
</tr>
<tr>
<td>Signed Word (W)</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>Double Word (UD)</td>
<td>[0, (2^{32}-1)]</td>
</tr>
<tr>
<td>Signed Double (D)</td>
<td>([-2^{31}, 2^{31}-1])</td>
</tr>
</tbody>
</table>

30  Reserved
29  Reserved: MBZ

28  **AccWrCtrl.** This field allows per instruction accumulator write control.

- 0 = don’t write result into accumulator
- 1 = AccWrCtrl. write result into accumulator, and destination

27:24  **CondModifier** or **CurrDst.RegNum[3:0]**

Definition of this bit field depends on whether the instruction is a *send/math* or not.

<table>
<thead>
<tr>
<th>Opcode != <em>send</em></th>
<th>Opcode = <em>send</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CondModifier:</strong></td>
<td><strong>CurrDst.RegNum[3:0]</strong></td>
</tr>
<tr>
<td>This field sets the flag register based on the internal conditional signals output from the execution pipe.</td>
<td>(See Instruction Reference chapter for CurrDst.)</td>
</tr>
</tbody>
</table>

23:21  **ExecSize - Execution Size.** This field determines the number of channels operating in parallel for this instruction. The size cannot exceed the maximum number of channels allowed for the given data type.

- 000b = 1 channel (scalar operation)
- 001b = 2 channels
- 010b = 4 channels
- 011b = 8 channels
- 100b = 16 channels
- 101 = 32 channels
- 110-111 = Reserved

20  **PredInv - Predicate Inverse.** This field, together with PredCtrl, enables and controls the generation of the predication mask for the instruction. When it is set, the predication uses the inverse of the predication bits.
Bits | Description
---|---
generated according to setting of Predicate Control. In other words, effect of PredInv happens after PredCtrl. This field is ignored by hardware if Predicate Control is set to 0000 - there is no predication.
0 = +. Positive polarity of predication.
1 = -. Negative polarity of predication.

19:16 **PredCtrl - Predicate Control.** This field, together with PredInv, enables and controls the generation of the predication mask for the instruction. It allows per-channel conditional execution of the instruction based on the content of the selected flag register. Encoding depends on the access mode.

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\|all operations. The same configuration is repeated for each group-of-4 execution channels.

See the **Predication** section for more information about predication.

In **Align1** access mode, there are twelve encodings (including no predication). The encodings applies to all execution channels with explicit channel grouping from single channel up to group of 16 channels.

**Predicate Control in Align16 access mode**
- 0000 = No predication (normal)
- 0001 = Predication with sequential flag channel mapping
- 0010 = Predication with replication swizzle \(x\)
- 0011 = Predication with replication swizzle \(y\)
- 0100 = Predication with replication swizzle \(z\)
- 0101 = Predication with replication swizzle \(w\)
- 0110 = Predication with \(\text{any}4h\)
- 0111 = Predication with \(\text{all}4h\)
- 1000 -1111 = Reserved

**Predicate Control in Align1 access mode**
- 0000 = No predication (normal)
- 0001 = Predication with sequential flag channel mapping
- 0010 = Predication with \(\text{any}v\) (any from f0.0-f0.1 on the same channel)
- 0011 = Predication with \(\text{all}v\) (all of f0.0-f0.1 on the same channel)
- 0100 = Predication with \(\text{any}2h\) (any in group of 2 channels)
- 0101 = Predication with \(\text{all}2h\) (all in group of 2 channels)
- 0110 = Predication with \(\text{any}4h\) (any in group of 4 channels)
- 0111 = Predication with \(\text{all}4h\) (all in group of 4 channels)
- 1000 = Predication with \(\text{any}8h\) (any in group of 8 channels)
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 = Predication with .all8h (all in group of 8 channels)</td>
<td></td>
</tr>
<tr>
<td>1010 = Predication with .any16h (any in group of 16 channels)</td>
<td></td>
</tr>
<tr>
<td>1011 = Predication with .all16h (all in group of 16 channels)</td>
<td></td>
</tr>
<tr>
<td>1100 = Predication with .any32h (any in group of 32 channels)</td>
<td></td>
</tr>
<tr>
<td>1101 = Predication with .all32h (all in group of 32 channels)</td>
<td></td>
</tr>
<tr>
<td>1110 -1111 = Reserved</td>
<td></td>
</tr>
</tbody>
</table>

15:14
ThreadCtrl - Thread Control. This field provides explicit control for thread switching.

- If this field is set to 00b, it is up to the GEN execution units to manage thread switching. This is the normal (and unnamed) 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. In another example, if the current instruction is ready to go, however, there is another thread with higher priority that also has an instruction ready, the EU switches to that thread.

- If this field is set to Switch, a forced thread switch occurs after the current instruction is executed and before the next instruction. In addition, a long delay (longer than the execution pipe latency) is introduced for the current thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. Switch is designed primarily as a safety feature in case there are race conditions for certain instructions.

- If this field is set to Atomic, the next instruction gets highest priority in thread arbitration for the execution pipeline.

  00b = Normal thread control
  10b = Switch
  01b = Atomic
  11b = Reserved

13:12
QtrCtrl - Quarter Control. This field provides explicit control for ARF selection.

This field combines with ExecSize determines which channels are used for the ARF registers.

Along with NibCtrl in DW1, 1/8 DMask/VMask and ARF can be selected.

<table>
<thead>
<tr>
<th>QtrCtrl</th>
<th>NibCtrl</th>
<th>ExecSize</th>
<th>Description</th>
<th>BNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>8</td>
<td>use first quarter for DMask/VMask use first half for everything else</td>
<td>1Q</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>8</td>
<td>use second quarter for DMask/VMask use second half for everything else</td>
<td>2Q</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>8</td>
<td>use third quarter for DMask/VMask use first half for everything else</td>
<td>3Q</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>8</td>
<td>use forth quarter for DMask/VMask use second half for everything else</td>
<td>4Q</td>
</tr>
<tr>
<td>Bits</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>use first half for DMask/VMask use all channels for everything else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>use second half for DMask/VMask use all channels for everything else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>use first 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>use second 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>use third 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>use fourth 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>use fifth 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>use sixth 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>use seventh 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>use eighth 1/8 for DMask/VMask and ARF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2H is only allowed for SIMD16 instruction in Single Program Flow mode (SPF=1).

NibCtrl is only allowed for SIMD4 instructions with (DF) double precision source and/or destination.

11:10

**DepCtrl - Destination Dependency Control.** This field selectively disabled destination dependency check and clear for this instruction.

When it is set to 00, normal destination dependency control is performed for the instruction - hardware checks for destination hazards to ensure data integrity. Specifically, destination register dependency check is conducted before the instruction is made ready for execution. After the instruction is executed, the destination register scoreboard will be cleared when the destination operands retire.

When bit 10 is set (**NoDDClr**), the destination register scoreboard will NOT be cleared when the destination operands retire. When bit 11 is set (**NoDDChk**), hardware does not check for destination register dependency before the instruction is made ready for execution. **NoDDClr** and **NoDDChk** are not mutual exclusive.

When this field is not all-zero, hardware does not protect against destination hazards for the instruction. This is typically used to assemble data in a fine grained fashion (e.g. matrix-vector compute with dot-product instructions), where the data integrity is guaranteed by software based on the intended usage of instruction sequences.

00 = Destination dependency checked and cleared (normal)
01 = **NoDDClr**. Destination dependency checked but not cleared
10 = **NoDDChk**. Destination dependency not checked but cleared
11 = **NoDDClr**, **NoDDChk**. Destination dependency not checked and not cleared

9

**MaskCtrl - Mask Control** (formerly Write Enable Control). This field determines if the per channel write enables are used to generate the final write enable. This field should be normally 0.

0 = use normal write enables (normal)
Bits | Description
--- | ---
1 = write all channels, except channels killed with predication control. ChanEn is ignored in this case. MaskCtrl = NoMask skips the check for PcIP[n] == ExIP before enabling a channel, as described in the Evaluate Write Enable section.

8 | **AccessMode - Access Mode.** This field determines the operand access for the instruction. It applies to all source and destination operands.
   
   When it is cleared (**Align1**), the instruction uses byte-aligned addressing for source and destination operands. Source swizzle control and destination mask control are not supported.
   
   When it is set (**Align16**), the instruction uses 16-byte-aligned addressing for all source and destination operands. Source swizzle control and destination mask control are supported in this mode.
   
   0 = **Align1**
   
   1 = **Align16**

7 | Reserved: MBZ (for future opcode extension)

6:0 | **Opcode - Instruction Operation Code.** This field contains the instruction operation code. Each opcode is given a unique mnemonic. For example, opcode 0x01 is for a move operation. Mnemonic for this opcode is **mov**.
   
   See section 5.3 for details of opcode encoding.

---

**Instruction Destination Doubleword (DW1)**

**DW1 1-src and 2-src Instructions**

Destination Doubleword (DW1) contains the register file and numeric type of all operands, as well as the register region parameters of the destination operand. See the Region Parameters section and the sections following it for more information about those parameters.

**Instruction Destination Doubleword**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:16 | **Destination Register Region.** This word contains the parameters describing the register region of the destination operand. Subfield definition depends on the AccessMode.
   
   See the Region Parameters section and the sections following it for more information about these parameters.

   **Programming Notes:**

   Although **Dst.HorzStride** is a don’t care for Align16, HW needs this to be programmed as 01.

| 15 | Reserved: MBZ |
### Bits

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:12</td>
<td><strong>Src1.SrcType – Source 1 Data Type.</strong> This field specifies the numeric data type of the source operand src1. The bits of a source operand are interpreted as the identified numeric data type, rather than coerced into a type implied by the operator. Depending on RegFile field of the source operand, there are two different encoding for this field. If a source is a register operand, this field follows the Source Register Type Encoding. If a source is an immediate operand, this field follows the Source Immediate Type Encoding. Source Register Type Encoding is identical to that for Destination Type. Source Immediate Type Encoding differs in two areas. First, it does not support byte and unsigned numeric data types. Second, it has three packed vector types, the V, UV, and VF types. <em>Implementation Note 1</em>: Both source operands, src0 and src1, support immediate types, but only one immediate is allowed for a given instruction and it must be the last operand. <em>Implementation Note 2</em>: Halfbyte integer vector (v) type can only be used in instructions in packed-word execution mode. Therefore, in a two-source instruction where src1 is of type :v, src0 must be of type :b, :ub, :w, or :uw.</td>
</tr>
</tbody>
</table>
| 10:3 | Source Register Type Encoding:  
000 = **UD. Unsigned Doubleword integer**  
001 = **D. Signed Doubleword integer**  
010 = **UW. Unsigned Word integer**  
011 = **W. Signed Word integer**  
100 = **UB. Unsigned Byte integer**  
101 = **B. Signed Byte integer**  
110 = **DF. Double precision Float (64-bit)** [DevIVB+]  
111 = **F. Single precision Float (32-bit)**  
Source Immediate Type Encoding:  
000 = **UD**  
001 = **D**  
010 = **UW**  
011 = **W**  
100 = **UV. 32-bit halfbyte Unsigned Integer Vector**  
101 = **VF. 32-bit restricted Vector Float**  
110 = **V. 32-bit halfbyte integer Vector**  
111 = **F** |
<p>| 11:10 | <strong>Src1.RegFile – Source 1 Register File.</strong> This field identifies the register file of source operand src1. |</p>
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 = ARF. Architecture Register File (a#, acc#, f#, n#, null, ip, etc.)</td>
<td></td>
</tr>
<tr>
<td>01 = GRF. General Register File (r#)</td>
<td></td>
</tr>
<tr>
<td>10 = <strong>Reserved</strong>. Do not use this encoding.</td>
<td></td>
</tr>
<tr>
<td>11 = IMM. Immediate</td>
<td></td>
</tr>
</tbody>
</table>

9:7 **Src0.SrcType – Source 0 Data Type.** This field is the *SrcType* for src0 operand. It has the same definitions as *Src1.SrcType*.

6:5 **Src0.RegFile – Source 0 Register File.** This field is the *RegFile* for src0 operand. It has the same definitions as *Src1.RegFile*.

4:2 **Dst.DstType – Destination Data Type.** This field specifies the numeric data type of the destination operand dst. The bits of the destination operand are interpreted as the identified numeric data type, rather than coerced into a type implied by the operator. For a *send* instruction, this field applies to the **CurrDst** – the current destination operand.

   Encoding:
   000 = **UD**. Unsigned Doubleword integer
   001 = **D**. Signed Doubleword integer
   010 = **UW**. Unsigned Word integer
   011 = **W**. Signed Word integer
   100 = **UB**. Unsigned Byte integer
   101 = **B**. Signed Byte integer
   110 = ["**DF**"] Double Precision Float (64-bit) [DevIVB+]
   111 = **F**. Single precision Float (32-bit)

1:0 **Dst.RegFile – Destination Register File.** This field identifies the register file of the destination operand dst. Note that it is obvious that immediate cannot be a destination operand.

   For a *send* instruction, this field applies to the **PostDst** – the post destination operand.

   Encoding:
   00 = ARF. Architecture Register File (a#, acc#, f#, n#, null, ip, etc.)
   01 = GRF. General Register File (r#)
   10 = **Reserved**. Do not use this encoding.
   11 = **reserved**

The following tables describe the Destination Register Region based on the access mode and addressing mode.
### Destination Register Region in Direct + Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><strong>Dst.AddrMode – Destination Address Mode.</strong> This field is the AddrMode for the destination operand. For a <strong>send</strong> instruction, this field applies to <strong>PostDst</strong> – the post destination operand. Addressing mode for <strong>CurrDst</strong> (current destination operand) is fixed as Direct. (See Instruction Reference chapter for <strong>CurrDst</strong> and <strong>PostDst</strong>.)</td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>12:5</td>
<td><strong>Dst.RegNum – Destination Register Number.</strong> This field is the RegNum field for the destination operand. For a <strong>send</strong> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Dst.SubRegNum[4].</strong> This is the 16-byte aligned sub-register address. For a <strong>send</strong> instruction, this field applies to <strong>CurrDst</strong>.</td>
</tr>
<tr>
<td>3:0</td>
<td><strong>Dst.ChanEn – Destination Channel Enable.</strong> The channel enable field for the destination operand. For a <strong>send</strong> instruction, this field applies to the <strong>CurrDst</strong>.</td>
</tr>
</tbody>
</table>

### Destination Register Region in Direct+Align1 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><strong>Dst.AddrMode – Destination Address Mode.</strong> This field is the AddrMode for the destination operand. For a <strong>send</strong> instruction, it applies to <strong>PostDst</strong>. Addressing mode for <strong>CurrDst</strong> is fixed as Direct.</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Dst.HorzStride – Destination Horizontal Stride.</strong> This field is the HorzStride for the destination operand. For a <strong>send</strong> instruction, this field applies to <strong>CurrDst</strong>. <strong>PostDst</strong> only uses the register number.</td>
</tr>
<tr>
<td>12:5</td>
<td><strong>Dst.RegNum – Destination Register Number.</strong> This field is the RegNum field for the destination operand. For a <strong>send</strong> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
<tr>
<td>4:0</td>
<td><strong>Dst.SubRegNum – Destination Sub-Register Number.</strong> This field is the SubRegNum for the destination operand.) Note: The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field. For a <strong>send</strong> instruction, this field applies to <strong>CurrDst</strong>.</td>
</tr>
</tbody>
</table>

### Destination Register Region in Indirect+Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1032</td>
<td></td>
</tr>
</tbody>
</table>
### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><strong>Dst.AddrMode</strong> – Destination Address Mode. This field is the AddrMode for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>. Addressing mode for <strong>CurrDst</strong> is fixed as Direct.</td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>12:10</td>
<td><strong>Dst.AddrSubRegNum</strong> – Destination Address Sub-Register Number. This field is the AddrSubRegNum for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
<tr>
<td>9:4</td>
<td><strong>Dst.AddrImm[9:4]</strong> This is the half-register aligned AddrImm field for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
<tr>
<td>3:0</td>
<td><strong>Dst.ChanEn</strong> – Destination Channel Enable. The channel enable field for the destination operand. For a <em>send</em> instruction, this field applies to <strong>CurrDst</strong>.</td>
</tr>
</tbody>
</table>

### Destination Register Region in Indirect+Align1 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><strong>Dst.AddrMode</strong> – Destination Address Mode. This field is the AddrMode for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>. Addressing mode for <strong>CurrDst</strong> is fixed as Direct.</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Dst.HorzStride</strong> – Destination Horizontal Stride This field is the HorzStride for the destination operand. For a <em>send</em> instruction, this field applies to <strong>CurrDst</strong>. <strong>PostDst</strong> only uses the register number.</td>
</tr>
<tr>
<td>12:10</td>
<td><strong>Dst.AddrSubRegNum</strong> – Destination Address Sub-Register Number. This field is the AddrSubRegNum for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
<tr>
<td>9:0</td>
<td><strong>Dst.AddrImm</strong> – Destination Address Immediate. This field is the byte-aligned AddrImm for the destination operand. For a <em>send</em> instruction, this field applies to <strong>PostDst</strong>.</td>
</tr>
</tbody>
</table>

### DW1 3-src Instructions

This section describes the field in DW1 for the 3-src instruction format.
### Instruction DW1

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td><strong>Destination Register Number.</strong> This field contains the destination register number.</td>
</tr>
<tr>
<td>23:21</td>
<td><strong>Destination Subregister Number.</strong> This field contains the destination sub-register number. <strong>Note:</strong> The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.</td>
</tr>
</tbody>
</table>
| 20:17  | **Destination Channel Enable.** Four channel enables are defined for controlling which channels are written into the destination region. These channel mask bits are applied in a modulo-four manner to all ExecSize channels. There is 1-bit Channel Enable for each channel within the group of 4. If the bit is cleared, the write for the corresponding channel is disabled. If the bit is set, the write is enabled. Mnemonics for the bit being set for the group of 4 are $x$, $y$, $z$, and $w$, respectively, where $x$ corresponds to Channel 0 in the group and $w$ corresponds to channel 3 in the group.  
0: Write Disabled  
1: Write Enabled (normal) |
| 16:15  | **Dst Type.** This field contains the data type for the destination.  
00b = Single Precision Float  
01b = DWord  
10b = Unsigned DWord  
11b = Double Precision Float |
| 14:13  | **Src Type.** This field contains the data type for all three sources.  
00b = Single Precision Float  
01b = DWord  
10b = Unsigned DWord  
11b = Double Precision Float |
| 12:10  | Reserved: MBZ |
| 9:8    | **Source2 Modifier.** This field contains the modifier for source2.  
Refer to Table 5-5 for the encoding. |
| 7:6    | **Source1 Modifier.** This field contains the modifier for source1.  
Refer to Table 5-5 for the encoding. |
| 5:4    | **Source0 Modifier.** This field contains the modifier for source0.  
Refer to Table 5-5 for the encoding. |
### Instruction Source 0 Doubleword 2 (DW2)

#### DW2 1-snc and 2-snc Instructions

Instruction Source 0 Doubleword 2 (DW2) contains the first source operand and also flag register number.

- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Direct Addressing with Align16.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Direct Addressing with Align1.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Indirect Addressing with Align16.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Indirect Addressing with Align1.

#### Instruction Source 0 Doubleword in Direct+Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>25</td>
<td><strong>FlagSubRegNum – Flag Sub-Register Number.</strong> This field specifies the sub-register number for a flag register operand. There are two sub-registers in the flag register. Each sub-register contains 16 flag bits. The selected flag sub-register is the source for predication if predication is enabled for the instruction. It is the destination to store conditional flag bits if conditional modifier is enabled for the instruction. The same flag sub-register can be both the predication source and conditional destination, if both predication and conditional modifier are enabled.</td>
</tr>
<tr>
<td>24:21</td>
<td><strong>Src0.VertStride – Source 0 Vertical Stride.</strong> This field is the VertStride for src0 operand. It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>2</td>
<td><strong>[DevIVB+] Flag Register Number.</strong> This field contains the flag register number for instructions with a non-zero Conditional Modifier.</td>
</tr>
<tr>
<td>1</td>
<td><strong>Flag Subregister Number.</strong> This field contains the flag sub-register number for instructions with a non-zero Conditional Modifier.</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 19:16     | **Src0.ChanSel[7:4]**    
This is bits [7:4] of the *ChanSel* field for src0 operand.                 |
| 15        | **Src0.AddrMode – Source 0 Address Mode.** This field is the *AddrMode* for src0 operand.  
It is ignored if src0 is an immediate operand. |
| 14:13     | **Src0.SrcMod – Source 0 Source Modifier.** This field is the *SrcMod* for source operand src0. |
| 12:5      | **Src0.RegNum – Source 0 Register Number**  
This is the *RegNum* field for source operand src0.  
It is ignored if src0 is an immediate operand. |
| 4         | **Src0.SubRegNum[4]**  
This is the 16-byte aligned sub-register address for source operand src0.  
It is ignored if src0 is an immediate operand.  
**Note:** The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field. For example, using the F (Float) type the possible sub-register numbers in Align16 mode are 0 or 4, corresponding to 0 or 1 for this field. |
| 3:0       | **Src0.ChanEn – Source 0 Channel Enable**  
This is the *ChanEn* field for source operand src0.  
It is ignored if src0 is an immediate operand. |

**Instruction Source 0 Doubleword in Direct+Align1 mode**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td><strong>Reserved: MBZ</strong></td>
</tr>
<tr>
<td>25</td>
<td><strong>FlagSubRegNum – Flag Sub-Register Number.</strong> This field specifies the sub-register number for a flag register operand.</td>
</tr>
</tbody>
</table>
| 24:21     | **Src0.VertStride – Source 0 Vertical Stride**  
This is the *VertStride* field for src0 operand.  
It is ignored if src0 is an immediate operand. |
| 20:18     | **Src0.Width.** This is the *Width* field for source operand src0.  
It is ignored if src0 is an immediate operand. |
### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17:16</td>
<td><strong>Src0.HorzStride.</strong> This is the <em>HorzStride</em> field for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>15</td>
<td><strong>Src0.AddrMode – Source 0 Address Mode.</strong> This is the <em>AddrMode</em> for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Src0.SrcMod – Source 0 Source Modifier.</strong> This is the <em>SrcMod</em> field for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>12:5</td>
<td><strong>Src0.RegNum – Source 0 Register Number.</strong> This is the <em>RegNum</em> field for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>4:0</td>
<td><strong>Src0.SubRegNum – Source 0 Sub-Register Number.</strong> This is the <em>SubRegNum</em> field for src0 operand.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.</td>
</tr>
</tbody>
</table>

### Instruction Source 0 Doubleword in Indirect+Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>25</td>
<td><strong>FlagSubRegNum – Flag Sub-Register Number.</strong> This field specifies the sub-register number for a flag register operand.</td>
</tr>
<tr>
<td>24:21</td>
<td><strong>Src0.VertStride – Source 0 Vertical Stride.</strong> This is the <em>VertStride</em> field for src0 operand.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>19:16</td>
<td><strong>Src0.ChanSel[7:4] – Source 0 Channel Select.</strong> This is bits [7:4] of the <em>ChanSel</em> field for src0 operand.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>15</td>
<td><strong>Src0.AddrMode – Source 0 Address Mode.</strong> This is the <em>AddrMode</em> for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Src0.SrcMod – Source 0 Source Modifier.</strong> This is the <em>SrcMod</em> field for source operand src0.</td>
</tr>
<tr>
<td></td>
<td>It is ignored if src0 is an immediate operand.</td>
</tr>
</tbody>
</table>
### Instruction Source 0 Doubleword in Indirect+Align1 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>25</td>
<td><strong>FlagSubRegNum – Flag Sub-Register Number.</strong> This field specifies the sub-register number for a flag register operand.</td>
</tr>
<tr>
<td>24:21</td>
<td><strong>Src0.VertStride – Source 0 Vertical Stride.</strong> This is the VertStride field for src0 operand.</td>
</tr>
<tr>
<td>20:18</td>
<td><strong>Src0.Width.</strong> This is the Width field for source operand src0.</td>
</tr>
<tr>
<td>17:16</td>
<td><strong>Src0.HorzStride.</strong> This is the HorzStride field for source operand src0.</td>
</tr>
<tr>
<td>15</td>
<td><strong>Src0.AddrMode – Source 0 Address Mode.</strong> This is the AddrMode for source operand src0.</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Src0.SrcMod – Source 0 Source Modifier.</strong> This is the SrcMod field for source operand src0.</td>
</tr>
<tr>
<td>12:10</td>
<td><strong>Src0.AddrSubRegNum – Source 0 Address Sub-Register Number.</strong> This is the AddrSubRegNum field for source operand src0.</td>
</tr>
<tr>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>9:0</td>
<td><strong>Src0.AddrImm – Source 0 Address Immediate.</strong> This is the byte aligned <code>AddrImm</code> field for src0. It is ignored if src0 is an immediate operand.</td>
</tr>
</tbody>
</table>

This section describes the field in DW2 and DW3 of the 3-src instruction format.

**Instruction DW2 and DW3 3-Source**

<table>
<thead>
<tr>
<th>DW</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW3</td>
<td>31:30</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>29:22</td>
<td><strong>Source2 Register Number.</strong> This field contains the register number for source2.</td>
</tr>
<tr>
<td></td>
<td>21:19</td>
<td><strong>Source2 Subregister Number.</strong> This field contains the sub-register number for source2. <strong>Note:</strong> The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.</td>
</tr>
<tr>
<td></td>
<td>18:11</td>
<td><strong>Source2 Channel Select.</strong> This field contains the swizzle control for source2. See ChanSel in the <a href="#">Common Instruction Fields</a> section for a description of the Source Swizzle encodings.</td>
</tr>
<tr>
<td></td>
<td>10:10</td>
<td><strong>Source2 Replication Control.</strong> This field controls replication for source2. See RepCtrl in the <a href="#">Common Instruction Fields</a> section for a description of the Source Replication Control encodings.</td>
</tr>
<tr>
<td></td>
<td>9:9</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td></td>
<td>8:1</td>
<td><strong>Source1 Register Number.</strong> This field contains the register number for source1.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td><strong>Source1 Subregister Number.</strong> This field contains the sub-register number for source1. <strong>Note:</strong> The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.</td>
</tr>
<tr>
<td>DW2</td>
<td>31:30</td>
<td><strong>Source1 Subregister Number.</strong> This field contains the sub-register number for source1. <strong>Note:</strong> The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.</td>
</tr>
<tr>
<td></td>
<td>29:22</td>
<td><strong>Source1 Channel Select.</strong> This field contains the swizzle control for source1. See ChanSel in the <a href="#">Common Instruction Fields</a> section for a description of the Source Swizzle encodings.</td>
</tr>
</tbody>
</table>
### Instruction Source 1 Doubleword 3 (DW3)

Instruction Source 1 Doubleword 3 (DW3) contains the second source operand (src1) and is used to hold the 32-bit immediate source (imm32 as src0 or src1). *Instruction Source 1 Doubleword 3 (DW3)* and *Instruction Source 1 Doubleword 3 (DW3)* define the fields in this doubleword with the following exceptions:

- If src0 is an immediate operand, this doubleword contains **imm32** for src0.
- If src1 is an immediate operand, this doubleword contains **imm32** for src1.
- If the instruction is a send, bit 31 of this doubleword contains **EOT** field.
  - If src1 is immediate, the remaining 31 bits in this doubleword is **MsgDesct31**.
  - If src1 is a register, src1 must be a0.0. The rest of this doubleword will be configured accordingly.
- If indirect address is supported for src1, *Instruction Source 1 Doubleword 3 (DW3)* and *Instruction Source 1 Doubleword 3 (DW3)* define the fields in DW3 for indirectly addressed src1 in Align16 and Align1 modes.
### Instruction Source 1 Doubleword in Direct + Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>
| 24:21 | **Src1.VertStride – Source 1 Vertical Stride.** This field is the VertStride for src1 operand.  
It is ignored if src1 is an immediate operand. |
| 20    | Reserved: MBZ                                                 |
| 19:16 | **Src1.ChanSel[7:4]**  
This contains bits [7:6] of the ChanSel field for src1 operand.  
It is ignored if src1 is an immediate operand. |
| 15    | Reserved: MBZ                                                 |
| 14:13 | **Src1.SrcMod – Source 1 Source Modifier.** This field is the SrcMod for src1 operand.  
It is ignored if src1 is an immediate operand. |
| 12:5  | **Src1.RegNum.**  
This field is the RegNum field for src1 operand.  
It is ignored if src1 is an immediate operand. |
| 4     | **Src1.SubRegNum[4].**  
This field is bit [4] of the SubRegNum field for src1.  
It is ignored if src1 is an immediate operand.  
**Note:** The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field. For example, using the F (Float) type the possible sub-register numbers in Align16 mode are 0 or 4, corresponding to 0 or 1 for this field. |
| 3:0   | **Src1.ChanEn – Source 1 Channel Enable.** It is the channel enable field for src1. It is ignored if src1 is an immediate operand. |
### Instruction Source 1 Doubleword in Direct + Align1 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>24:21</td>
<td><strong>Src1.VertStride – Source 1 Vertical Stride.</strong> This field is the VertStride for src1 operand. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>20:18</td>
<td><strong>Src1.Width.</strong> This is the Width field for source operand src1. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>17:16</td>
<td><strong>Src1.HorzStride.</strong> This is the HorzStride field for source operand src1. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>14:13</td>
<td><strong>Src1.SrcMod – Source 1 Source Modifier.</strong> This field is the SrcMod for src1 operand. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>12:5</td>
<td><strong>Src1.RegNum – Source 1 Register Number.</strong> This is the RegNum field for source operand src1. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>4:0</td>
<td><strong>Src1.SubRegNum – Source 1 Sub-Register Number.</strong> This is the SubRegNum field for source operand src1. It is ignored if src1 is an immediate operand.</td>
</tr>
</tbody>
</table>

**Note:** The recommended instruction syntax uses GRF sub-register numbers in units of element size, which the assembler translates to the appropriate value for this field.

### Instruction Source 1 Doubleword in Indirect+Align16 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>24:21</td>
<td><strong>Src1.VertStride – Source 1 Vertical Stride</strong> This is the VertStride field for src1 operand. It is ignored if src1 is an immediate operand.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved: MBZ</td>
</tr>
<tr>
<td>19:16</td>
<td><strong>Src1.ChanSel[7:4] – Source 1 Channel Select</strong> This is bits [7:4] of the ChanSel field for src1 operand.</td>
</tr>
</tbody>
</table>
### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th><strong>Src1.AddrMode – Source 1 Address Mode</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the AddrMode for source operand src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14:13</th>
<th><strong>Src1.SrcMod – Source 1 Source Modifier</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the SrcMod field for source operand src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12:10</th>
<th><strong>Src1.AddrSubRegNum – Source 1 Address Sub-Register Number</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the AddrSubRegNum field for source operand src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9:4</th>
<th><strong>Src1.AddrImm[9:4] – Source 1 Address Immediate</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This contains the half-register aligned AddrImm field ([bits [9:4]) for src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3:0</th>
<th><strong>Src1.ChanEn – Source 1 Channel Enable</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the ChanEn field for source operand src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Source 1 Doubleword in Indirect+Align1 mode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>Reserved: MBZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24:21</th>
<th><strong>Src1.VertStride – Source 1 Vertical Stride</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the VertStride field for src1 operand.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20:18</th>
<th><strong>Src1.Width</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the Width field for source operand src1.</td>
<td></td>
</tr>
<tr>
<td>It is ignored if src1 is an immediate operand.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17:16</th>
<th><strong>Src1.HorzStride</strong></th>
</tr>
</thead>
</table>
Bits | Description
--- | ---
| This is the *HorzStride* field for source operand src1. It is ignored if src1 is an immediate operand.
| 15 | **Src1.AddrMode – Source 1 Address Mode**
| This is the *AddrMode* field for source operand src1. It is ignored if src1 is an immediate operand.
| 14:13 | **Src1.SrcMod – Source 1 Source Modifier**
| This is the *SrcMod* field for source operand src1. It is ignored if src1 is an immediate operand.
| 12:10 | **Src1.AddrSubRegNum – Source 1 Address Sub-Register Number**
| This is the *AddrSubRegNum* field for source operand src1. It is ignored if src1 is an immediate operand.
| 9:0 | **Src1.AddrImm – Source 1 Address Immediate**
| This is the byte aligned *AddrImm* field for src1. It is ignored if src1 is an immediate operand.

**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.

In some flow control instructions, IP offsets, such as the JIP and UIP instruction fields, are measured in 64-bit QWords. Thus a compact 64-bit instruction is 1 unit for IP offset calculations and a native 128-bit instruction is 2 units for IP offset calculations. However other instructions use a new relative 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 DevHSW. For these processors, instructions with three source operands cannot be compacted.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Size</th>
<th>Mapping Bits</th>
<th>Compact Name</th>
<th>Mapping Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>55:48</td>
<td>8</td>
<td>76:69</td>
<td>Src0.RegNum</td>
<td>Src0.RegNum.</td>
</tr>
<tr>
<td>34:30</td>
<td>5</td>
<td>88:77</td>
<td>Src0Index</td>
<td>Lookup one of 32 12-bit values. That value is used (from MSB to LSB) for the Src0.AddrMode, Src0.ChanSel[7:4], Src0.HorzStride, Src0.SrcMod, Src0.VertStride, and Src0.Width bit fields. Note that this field spans a DWord boundary within the QWord compacted instruction.</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>29</td>
<td>CmptCtrl</td>
<td>Compaction Control. The same in both the compact and native formats: 0: Regular instruction, not compacted. 1: Compacted instruction.</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>[HSW]: Not mapped.</td>
<td>[HSW]: Reserved</td>
<td>[HSW]: Not mapped. MBZ.</td>
</tr>
<tr>
<td>27:24</td>
<td>4</td>
<td>27:24</td>
<td>CondModifier</td>
<td>CondModifier. The same in both the compact and native formats.</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>28</td>
<td>AccWrCtrl</td>
<td>AccWrCtrl.</td>
</tr>
<tr>
<td>22:18</td>
<td>5</td>
<td>100:96, 68:64, 52:48</td>
<td>SubRegIndex</td>
<td>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.</td>
</tr>
<tr>
<td>17:13</td>
<td>5</td>
<td>63:61, 46:32</td>
<td>DataTypeIndex</td>
<td>Lookup one of 32 18-bit values. That value is used (from MSB to LSB) for the Dst.AddrMode, Dst.HorzStride, Dst.DstType, Dst.RegFile, Src0.SrcType, Src0.RegFile, Src1.SrcType, and Src1.RegType bit fields.</td>
</tr>
<tr>
<td>12:8</td>
<td>5</td>
<td>[HSW]: 90:89, 31, 23:8</td>
<td>ControlIndex</td>
<td>HSW: 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.</td>
</tr>
<tr>
<td>6:0</td>
<td>7</td>
<td>6:0</td>
<td>Opcode</td>
<td>Opcode. The same in both the compact and native formats.</td>
</tr>
</tbody>
</table>
The following diagram is an alternate presentation of the [HSW] compact instruction format.

**GEN Compact Instruction Format**

**EU Instruction Compaction Tables**

The following four tables describe the mappings for the ControlIndex, DataTypeIndex, SubRegIndex, Src0Index, and Src1Index fields in the compact instruction format for HSW.

<table>
<thead>
<tr>
<th>ControlIndex</th>
<th>19-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000000000010</td>
<td>Align1</td>
</tr>
<tr>
<td>1</td>
<td>000010000000000000</td>
<td>Align1</td>
</tr>
<tr>
<td>2</td>
<td>000010000000000001</td>
<td>Align16</td>
</tr>
<tr>
<td>3</td>
<td>000010000000000010</td>
<td>Align1</td>
</tr>
<tr>
<td>4</td>
<td>000010000000000011</td>
<td>Align16</td>
</tr>
<tr>
<td>5</td>
<td>000010000000000100</td>
<td>Align1</td>
</tr>
<tr>
<td>6</td>
<td>000010000000000101</td>
<td>Align16</td>
</tr>
<tr>
<td>7</td>
<td>000010000000000111</td>
<td>Align16</td>
</tr>
<tr>
<td>8</td>
<td>000010000000001000</td>
<td>Align1</td>
</tr>
<tr>
<td>9</td>
<td>000010000000001001</td>
<td>Align16</td>
</tr>
<tr>
<td>10</td>
<td>000010000000001100</td>
<td>Align16</td>
</tr>
<tr>
<td>11</td>
<td>000011000000000000</td>
<td>Align1</td>
</tr>
<tr>
<td>12</td>
<td>000011000000000001</td>
<td>Align16</td>
</tr>
<tr>
<td>13</td>
<td>000011000000000010</td>
<td>Align1</td>
</tr>
<tr>
<td>14</td>
<td>000011000000000011</td>
<td>Align16</td>
</tr>
<tr>
<td>15</td>
<td>000011000000000100</td>
<td>Align1</td>
</tr>
<tr>
<td>16</td>
<td>000011000000000101</td>
<td>Align16</td>
</tr>
</tbody>
</table>
### Control Index

<table>
<thead>
<tr>
<th>ControlIndex</th>
<th>19-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0000110000000000111</td>
<td>Align16</td>
</tr>
<tr>
<td>18</td>
<td>0000110000000000101</td>
<td>Align16</td>
</tr>
<tr>
<td>19</td>
<td>00001100000000001101</td>
<td>Align16</td>
</tr>
<tr>
<td>20</td>
<td>000011000000000010000</td>
<td>Align1</td>
</tr>
<tr>
<td>21</td>
<td>00001100000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>22</td>
<td>00010000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>23</td>
<td>00010000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>24</td>
<td>00010000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>25</td>
<td>00010000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>26</td>
<td>00101100000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>27</td>
<td>00101100000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>28</td>
<td>00110000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>29</td>
<td>00110000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>30</td>
<td>01010000000000001000000</td>
<td>Align1</td>
</tr>
<tr>
<td>31</td>
<td>01010000000000001000000</td>
<td>Align1</td>
</tr>
</tbody>
</table>

### Data Type Index

<table>
<thead>
<tr>
<th>Data Type Index</th>
<th>18-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001000000000000001</td>
<td>r:ud</td>
</tr>
<tr>
<td>1</td>
<td>001000000000000010</td>
<td>a:ud</td>
</tr>
<tr>
<td>2</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>3</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>4</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>5</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>6</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>7</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>8</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>9</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>10</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>11</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>12</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>13</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>14</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>15</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>16</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
<tr>
<td>17</td>
<td>001000000000000010</td>
<td>r:ud</td>
</tr>
</tbody>
</table>
### 18-Bit Mapping

<table>
<thead>
<tr>
<th>DataTypeIndex</th>
<th>18-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0011110111101101</td>
<td>`rf</td>
</tr>
<tr>
<td>19</td>
<td>0011111111111110</td>
<td>`af</td>
</tr>
<tr>
<td>20</td>
<td>0000000001000001100</td>
<td>`aw</td>
</tr>
<tr>
<td>21</td>
<td>0010000000000111101</td>
<td>`rf</td>
</tr>
<tr>
<td>22</td>
<td>0010000000010100101</td>
<td>`r:d</td>
</tr>
<tr>
<td>23</td>
<td>0010000100000000010000</td>
<td>`a:ud</td>
</tr>
<tr>
<td>24</td>
<td>0010010100101001000100</td>
<td>`a:d</td>
</tr>
<tr>
<td>25</td>
<td>00100111101000010000100</td>
<td>`r:d</td>
</tr>
<tr>
<td>26</td>
<td>0010100101000001000001001</td>
<td>`r:uw</td>
</tr>
<tr>
<td>27</td>
<td>00110111111011111111101</td>
<td>`r:f</td>
</tr>
<tr>
<td>28</td>
<td>00111111111111111111111</td>
<td>`r:f</td>
</tr>
<tr>
<td>29</td>
<td>0010111101110101001010000</td>
<td>`a:w</td>
</tr>
<tr>
<td>30</td>
<td>0010100101010101000010100</td>
<td>`a:uw</td>
</tr>
<tr>
<td>31</td>
<td>0010101101010101001010000</td>
<td>`auw</td>
</tr>
</tbody>
</table>

### 15-Bit Mapping

<table>
<thead>
<tr>
<th>SubRegIndex</th>
<th>15-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0000000000000001</td>
<td>0.x</td>
</tr>
<tr>
<td>2</td>
<td>0000000000010000</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0000000000001111</td>
<td>0.xyzw</td>
</tr>
<tr>
<td>4</td>
<td>0000000000010000</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>0000000010000000</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0000000100000000</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0000000110000000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0000010000000000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0000010000010000</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>0000010100000000</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0010000000000000</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0010000000000001</td>
<td>0.x</td>
</tr>
<tr>
<td>13</td>
<td>0010000000000001</td>
<td>0.x</td>
</tr>
<tr>
<td>14</td>
<td>0000000100000100</td>
<td>0.y</td>
</tr>
<tr>
<td>15</td>
<td>0000000100000111</td>
<td>0.x</td>
</tr>
<tr>
<td>16</td>
<td>0000000100000100</td>
<td>0.z</td>
</tr>
<tr>
<td>17</td>
<td>0000000100010011</td>
<td>0.xz</td>
</tr>
<tr>
<td>18</td>
<td>0000000100010000</td>
<td>0.w</td>
</tr>
</tbody>
</table>
### SubRegIndex or Src1Index Compact Instruction Field Mappings

<table>
<thead>
<tr>
<th>SubRegIndex</th>
<th>15-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>001000010001110</td>
<td>0.yzw</td>
</tr>
<tr>
<td>20</td>
<td>001000010001111</td>
<td>0.xyzw</td>
</tr>
<tr>
<td>21</td>
<td>0010001100000000</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0010001111010000</td>
<td>0.w</td>
</tr>
<tr>
<td>23</td>
<td>0100000000000000</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0100001100000000</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0110000000000000</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0111100100001111</td>
<td>0.xyz</td>
</tr>
<tr>
<td>27</td>
<td>1000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1010000000000000</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>1100000000000000</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1110000000000000</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>1110000000111100</td>
<td>28</td>
</tr>
</tbody>
</table>

### Src0Index or Src1Index Compact Instruction Field Mappings

<table>
<thead>
<tr>
<th>Src0Index or Src1Index</th>
<th>12-Bit Mapping</th>
<th>Mapped Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000000</td>
<td>dir</td>
</tr>
<tr>
<td>1</td>
<td>000000000010</td>
<td>(-)</td>
</tr>
<tr>
<td>2</td>
<td>000000010000</td>
<td>dir</td>
</tr>
<tr>
<td>3</td>
<td>000000100100</td>
<td>(-)</td>
</tr>
<tr>
<td>4</td>
<td>000000110000</td>
<td>dir</td>
</tr>
<tr>
<td>5</td>
<td>000000100000</td>
<td>dir</td>
</tr>
<tr>
<td>6</td>
<td>000000010100</td>
<td>dir</td>
</tr>
<tr>
<td>7</td>
<td>000001001000</td>
<td>dir</td>
</tr>
<tr>
<td>8</td>
<td>000001010000</td>
<td>dir</td>
</tr>
<tr>
<td>9</td>
<td>000001110000</td>
<td>dir</td>
</tr>
<tr>
<td>10</td>
<td>000001111000</td>
<td>dir</td>
</tr>
<tr>
<td>11</td>
<td>001100000000</td>
<td>dir</td>
</tr>
<tr>
<td>12</td>
<td>001100000010</td>
<td>(-)</td>
</tr>
<tr>
<td>13</td>
<td>001100010100</td>
<td>dir</td>
</tr>
<tr>
<td>14</td>
<td>001100010000</td>
<td>dir</td>
</tr>
<tr>
<td>15</td>
<td>001100010010</td>
<td>(-)</td>
</tr>
<tr>
<td>16</td>
<td>001100100000</td>
<td>dir</td>
</tr>
<tr>
<td>17</td>
<td>001100101000</td>
<td>dir</td>
</tr>
<tr>
<td>18</td>
<td>001100111000</td>
<td>dir</td>
</tr>
</tbody>
</table>
### Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

The opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 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 (\textit{cmp} and \textit{cmpn}) share the common 6 MSBs in the form of 001000xb. Bit 0 indicates whether it is a normal compare, \textit{cmp}, or a special compare-NaN, \textit{cmpn}.

### Move and Logic Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov</td>
<td>Component-wise move</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>sel</td>
<td>Component-wise selective move based on predication</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>movi</td>
<td>Fast component-wise indexed move</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>not</td>
<td>Component-wise one's complement (bitwise not)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>and</td>
<td>Component-wise logical AND (bitwise and)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>or</td>
<td>Component-wise logical OR (bitwise or)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>xor</td>
<td>Component-wise logical XOR (bitwise xor)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>shr</td>
<td>Component-wise logical shift right</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>shl</td>
<td>Component-wise logical shift left</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>dim</td>
<td>Double Precision Floating Point Immediate Data Move</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>asr</td>
<td>Component-wise arithmetic shift right</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cmp</td>
<td>Component-wise compare, store condition code in destination</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>cmpn</td>
<td>Component-wise compare-NaN, store condition code in destination</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>csel</td>
<td>Component-wise selective move based on result of compare</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>\textit{f32tof16}</td>
<td>Single precision float to half precision float conversion</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Opcode

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0x14</td>
<td><code>f16to32</code></td>
<td>Half precision float to single precision float conversion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0x15</td>
<td><code>Reserved</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0x16</td>
<td><code>Reserved</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0x17</td>
<td><code>brev</code></td>
<td>Reverse bits</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0x18</td>
<td><code>bfe</code></td>
<td>Bitfield exact</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0x19</td>
<td><code>bfi1</code></td>
<td>Bitfield insert macro instruction 1, generate mask</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>0x1A</td>
<td><code>bfi2</code></td>
<td>Bitfield insert macro instruction 2, generate mask</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>27-31</td>
<td>0x1B-0x1F</td>
<td><code>Reserved</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Flow Control Instructions**

This instruction group has an opcode format of 010xxxxb.

### Flow Control Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><code>jmp</code></td>
<td>Jump indexed</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td><code>brd</code></td>
<td>Branch - Diverging</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td><code>if</code></td>
<td>If</td>
<td>0/2</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td><code>brc</code></td>
<td>Branch - Converging</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>36</td>
<td><code>else</code></td>
<td>Else</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td><code>endif</code></td>
<td>End if</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction</td>
<td>Description</td>
<td>#src</td>
<td>#dst</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>38</td>
<td>case</td>
<td>Case – Inside Switch block</td>
<td>0/2</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>while</td>
<td>While</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>break</td>
<td>Break</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>cont</td>
<td>Continue</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>halt</td>
<td>Halt</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>calla</td>
<td>Subroutine call absolute</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>call</td>
<td>Subroutine call</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>return</td>
<td>Subroutine return</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Miscellaneous Instructions**

This instruction group has an opcode format of 011xxxxb.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>wait</td>
<td>Wait for (external) notification</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>send</td>
<td>Send</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>sendc</td>
<td>Conditional Send (based on TDR)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>53-55</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>math</td>
<td>Math functions for extended math pipeline</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>57-63</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 0x40</td>
<td>add</td>
<td>Component-wise addition</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>65 0x41</td>
<td>mul</td>
<td>Component-wise multiply</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>66 0x42</td>
<td>avg</td>
<td>Component-wise average of the two source operands</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>67 0x43</td>
<td>frc</td>
<td>Component-wise floating point truncate-to-minus-infinity fraction</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>68 0x44</td>
<td>rdu</td>
<td>Component-wise floating point rounding up (ceiling)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>69 0x45</td>
<td>rdd</td>
<td>Component-wise floating point rounding down (floor)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>70 0x46</td>
<td>rde</td>
<td>Component-wise floating point rounding toward nearest even</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>71 0x47</td>
<td>rndz</td>
<td>Component-wise floating point rounding toward zero</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>72 0x48</td>
<td>mac</td>
<td>Component-wise multiply accumulate</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>73 0x49</td>
<td>mach</td>
<td>multiply accumulate high</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>74 0x4A</td>
<td>lzd</td>
<td>leading zero detection</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>75 0x4B</td>
<td>fbb</td>
<td>Find first 1 for UD from msb side, or first 1/0 for D.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>76 0x4C</td>
<td>fbl</td>
<td>First first 1 for UD from lsb side</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>77 0x4D</td>
<td>cbi</td>
<td>Count bits set</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>78 0x4E</td>
<td>addc</td>
<td>Integer add with carry</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Vector Arithmetic Instructions

- This instruction group has an opcode format of 101xxxxb.

### Vector Arithmetic Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>hex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>0x4F</td>
<td>subb</td>
<td>integer subtract with borrow</td>
<td>2</td>
</tr>
<tr>
<td>75-79</td>
<td>0x4B-0x4F</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vector Arithmetic Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>hex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>0x50</td>
<td>sad2</td>
<td>2-wide sum of absolute difference</td>
<td>2</td>
</tr>
<tr>
<td>81</td>
<td>0x51</td>
<td>sada2</td>
<td>2-wide sad accumulate</td>
<td>2</td>
</tr>
<tr>
<td>82-83</td>
<td>0x52-0x53</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>0x54</td>
<td>dp4</td>
<td>4-wide dot product for 4-vector</td>
<td>2</td>
</tr>
<tr>
<td>85</td>
<td>0x55</td>
<td>dph</td>
<td>4-wide homogenous dot product for 4-vector</td>
<td>2</td>
</tr>
<tr>
<td>86</td>
<td>0x56</td>
<td>dp3</td>
<td>3-wide dot product for 4-vector</td>
<td>2</td>
</tr>
<tr>
<td>87</td>
<td>0x57</td>
<td>dp2</td>
<td>2-wide dot product for 4-vector</td>
<td>2</td>
</tr>
<tr>
<td>88</td>
<td>0x58</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>0x59</td>
<td>line</td>
<td>Component-wise line equation computation (a multiply-add)</td>
<td>2</td>
</tr>
<tr>
<td>90</td>
<td>0x5A</td>
<td>pln</td>
<td>Component-wise floating point plane equation computation (a multiply-multiply-add)</td>
<td>2</td>
</tr>
<tr>
<td>91</td>
<td>0x5B</td>
<td>fma(mad)</td>
<td>Component-wise floating point mad computation (a multiple-add)</td>
<td>3</td>
</tr>
<tr>
<td>92</td>
<td>0x5C</td>
<td>lrp</td>
<td>Component-wise floating point lrp computation (blend)</td>
<td>3</td>
</tr>
<tr>
<td>93</td>
<td>0x5D</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>94-95</td>
<td>0x5E-0x5F</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
<th>#src</th>
<th>#dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
<td>illegal</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>96-124</td>
<td>0x60-0x7D</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>0x7E</td>
<td>nop</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>127</td>
<td>0x7F</td>
<td>Reserved</td>
<td>(may be used as an extension code)</td>
<td></td>
</tr>
</tbody>
</table>
Native Instruction BNF

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

Instruction Groups

<Instruction>::= <UnaryInstruction>

| <BinaryAccInstruction> | <BinaryInstruction> | <TriInstruction> | <JumpInstruction> | <BranchLoopInstruction> | <ElseInstruction> | <BreakInstruction> | <MaskControlInstruction> | <TriInstruction2> | <CallInstruction> | <BranchConvIntruction> | <BranchDivInstruction> | <MathInstruction> | <SyncInstruction> | <SpecialInstruction>

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

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

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

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

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

<BinaryOp>::= mul | mac | mach | line | pln
| sad2 | sada2 | dp4 | dph | dp3 | dp2 | Irp | bfi1 | addc | subb

<BinaryAccInstruction>::= <Predicate> <BinaryAccInst> <ExecSize> dst <SrcAcc> <SrcImm> <InstrOptions>

<BinaryAccInst>::= <BinaryAccOp> <ConditionalModifier> <Saturate>

<BinaryAccOp>::= avg | add | sel
| and | or | xor
| shr | shl | asr
| cmp | cmpn |

<TriInstruction>::= <Predicate> <TriInst> <ExecSize> <PostDst> <CurrDst> <TriSrc> <MsgDesc> <InstOptions>

<TriInst>::= <TriOp> <ConditionalModifier> <Saturate>

<TriOp>::= send

<TriInstruction2> ::= <Predicate> <TriInst2> <ExecSize> dst <Src> <Src> <Src> <InstOptions>

<TriInst2>::= <TriOp> <ConditionalModifier> <Saturate>

<TriOp>::= bfe | bfi2 | mad

<BranchConvInstruction> ::= <Predicate> <BranchConvOp> <ExecSize> < RelativeLocation2>

<BranchConvOp>::= brc

<BranchConvInstruction> ::= <Predicate> <BranchDivOp> <ExecSize> < RelativeLocation3>

<BranchDivOp>::= brd

<CallInstruction> ::= <Predicate> <CallOp> <ExecSize> dst < RelativeLocation2>

<CallOp>::= call | CALLA

<MathInstruction> ::= <Predicate> <MathInst> <ExecSize> <Dst> <Src> <Src> <FC>

[MathInst]::= <MathOp> <Saturate>

[MathOp]::= math

<FC>::= INV | LOG | EXP | SQRT | RSQ | POW | SIN | COS | INT DIV

<JumpInstruction>::= <JumpOp> <RelativeLocation2>

<JumpOp>::= jmpi

<BranchLoopInstruction> ::= <Predicate> <BranchLoopOp> < RelativeLocation>

<BranchLoopOp>::= if | iff | while

<ElseInstruction>::= <ElseOp> < RelativeLocation>

<ElseOp>::= else

<BreakInstruction> ::= <Predicate> <BreakOp> <LocationStackCtrl>

<BreakOp>::= break | cont | halt

<SyncInstruction> ::= <Predicate> <SyncOp> <NotifyReg>

<SyncOp>::= wait

<SpecialInstruction>::= do | endif | nop | illegal
Source Register

Source with Accumulator Access and with Immediate

\(<\text{SrcAccImm}>::=\langle\text{SrcAcc}\rangle\)

|\(<\text{Imm32}\>\langle\text{SrcImmType}\rangle\)

\(<\text{SrcAcc}>::=\langle\text{DirectSrcAccOperand}\rangle\)

|\(<\text{IndirectSrcOperand}\rangle\)

\(<\text{DirectSrcAccOperand}>::=\langle\text{DirectSrcOperand}\rangle\)

|\(<\text{SrcArcOperandEx}\rangle\)

|\(<\text{AccReg}\>\langle\text{SrcType}\rangle\)

\(<\text{SrcArcOperandEx}>::=\langle\text{FlagReg}\>\langle\text{Region}\>\langle\text{SrcType}\rangle\)

|\(<\text{AddrReg}\>\langle\text{Region}\>\langle\text{SrcType}\rangle\)

|\(<\text{ControlReg}\rangle\)

|\(<\text{StateReg}\rangle\)

|\(<\text{NotifyReg}\rangle\)

|\(<\text{IPReg}\rangle\)

|\(<\text{NullReg}\rangle\)

|\(<\text{ChannelEnableReg}\rangle\)

|\(<\text{ThreadControlReg}\rangle\)

|\(<\text{PerformanceReg}\rangle\)

\(<\text{IndirectSrcOperand}>::=\langle\text{SrcModifier}\>\langle\text{IndirectGenReg}\>\langle\text{IndirectRegion}\>\langle\text{Swizzle}\>\langle\text{SrcType}\rangle\)

Source without Accumulator Access

\(<\text{Src}>::=\langle\text{DirectSrcOperand}\rangle\)

|\(<\text{IndirectSrcOperand}\rangle\)

\(<\text{DirectSrcOperand}>::=\langle\text{SrcModifier}\>\langle\text{DirectGenReg}\>\langle\text{Region}\>\langle\text{Swizzle}\>\langle\text{SrcType}\rangle\)

|\(<\text{SrcArcOperandEx}\rangle\)

\(<\text{TriSrc}>::=\langle\text{SrcModifier}\>\langle\text{DirectGenReg}\>\langle\text{Region}\>\langle\text{Swizzle}\>\langle\text{SrcType}\rangle\)

|\(<\text{NullReg}\rangle\)

\(<\text{MsgDesc}>::=\langle\text{ImmDesc}\rangle\)
|<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 sub-register 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 sub-register numbers 0 to 7, corresponding to sub-register 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
<SrcImmType> ::= <SrcType> | :v | :vf | :uv
<DstType> ::= <SrcType>

Write Mask

<WriteMask>::=
| . x | . y | . z | . w
| . xy | . xz | . xw | . yz | . yw | . zw
| . xyz | . xyw | . xzw | . yzw
| . xyzw

Swizzle Control

<Swizzle>::=
| . <ChanSel>
| . <ChanSel> <ChanSel> <ChanSel> <ChanSel>
<ChanSel>::= x | y | z | w

Immediate Values

<ImmAddrNum>::= -512...511
<Imm64> ::= 0.0...±1.0*2^-1024...1023 | 0...264-1 | -263...263-1
<Imm32> ::= 0.0...±1.0*2^-128...127 | 0...2^31-1 | -2^31...2^31-1
\[\text{Imm16} ::= 0... 2^{16{-}1} | -2^{15}... 2^{15{-}1}\]
\[\text{ImmDesc} ::= 0... 2^{32{-}1}\]

**Predication and Modifiers**

**Instruction Predication**

\[\text{<Predicate>} ::= ( \text{<PredState>} \text{<FlagReg>} \text{<PredCntrl>} )\]
\[\text{<PredState>} ::= + | -\]
\[\text{<PredCntrl>} ::= .x | .y | .z | .w | .any2h | .all2h | .any4h | .all4h | .any8h | .all8h | .any16h | .all16h | .anyv | .allv | .any32h | .all32h\]

**Source Modification**

\[\text{<SrcModifier>} ::= - | (\text{abs}) | - (\text{abs})\]

**Instruction Modification**

\[\text{<ConditionalModifier>} ::= (\text{<CondMod>}. \text{<FlagReg>}\]
\[\text{<CondMod>} ::= .z | .e| .nz | .ne| .g| .ge| .l| .le| .o | .r | .u\]
\[\text{<Saturate>} ::= .sat\]

**Execution Size**

\[\text{<ExecSize>} ::= (\text{<NumChannels>}\)

1064
<NumChannels>::= 1 | 2 | 4 | 8 | 16 | 32

**Instruction Options**

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

| **Atomic**

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

The columns in the following tables specify instruction mnemonics, hex opcodes, full names, instruction groups, processor generation (where blank means available for DevSNB+), the number of source operands, whether the instruction supports predication, any support for source modifiers, an indication of supported data types, whether the instruction supports saturation, and any support for conditional modifiers.

See the separate Accumulator Restrictions table for information about how instructions are allowed to use accumulators.

With a dozen columns in these tables, some terse notation is used, like IVB+ for DevIVB+ in the Gen (processor generation) column. If the Gen column is blank, an instruction is supported for DevSNB+, all generations implemented or designed so far, from Sandy Bridge forward.

N and Y indicate No (no support for a feature) and Yes (full support for a feature) respectively.

A SrcMod (source modifier) value of Y indicates that a numeric source modifier is allowed, optionally specifying absolute value, negation, or a forced negative value. The value N indicates no source modifier support.

A SrcMod value of ** indicates a numeric source modifier.

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.
### Instruction Set Summary Table A to B (Listed by Instruction Mnemonic)

<table>
<thead>
<tr>
<th>Mnem.</th>
<th>Hex Opcode</th>
<th>Name</th>
<th>Group</th>
<th>Gen</th>
<th>Srcs</th>
<th>Pred?</th>
<th>SrcMod</th>
<th>Src Types</th>
<th>Dst Type</th>
<th>Sat?</th>
<th>CondMod?</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>40</td>
<td>Addition</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>addc</td>
<td>4E</td>
<td>Integer Addition with Carry</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>N</td>
<td>UD</td>
<td>UD</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>05</td>
<td>Logic And</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>**</td>
<td>Int</td>
<td>Int</td>
<td>N</td>
<td>Y</td>
<td>Equality only</td>
</tr>
<tr>
<td>asr</td>
<td>12</td>
<td>Arithmetic Shift Right</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>Int</td>
<td>Int</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>avg</td>
<td>42</td>
<td>Average</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>B, UB</td>
<td>W, UW</td>
<td>D, UD</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>bfe</td>
<td>18</td>
<td>Bit Field Extract</td>
<td>Move and Logic</td>
<td>3</td>
<td>Y</td>
<td>N</td>
<td>UD, D</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>bfi1</td>
<td>19</td>
<td>Bit Field Insert 1</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>N</td>
<td>UD, D</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>bfi2</td>
<td>1A</td>
<td>Bit Field Insert 2</td>
<td>Move and Logic</td>
<td>3</td>
<td>Y</td>
<td>N</td>
<td>UD, D</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>bfrev</td>
<td>17</td>
<td>Bit Field Reverse</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>UD</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>brc</td>
<td>23</td>
<td>Branch Converging</td>
<td>Flow Control</td>
<td>0 or 1</td>
<td>Y</td>
<td>N</td>
<td>D</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>brd</td>
<td>21</td>
<td>Branch Diverging</td>
<td>Flow Control</td>
<td>0 or 1</td>
<td>Y</td>
<td>N</td>
<td>D</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>28</td>
<td>Break</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
### Instruction Set Summary Table C to E (Listed by Instruction Mnemonic)

<table>
<thead>
<tr>
<th>Mnem.</th>
<th>Hex Opcode</th>
<th>Name</th>
<th>Group</th>
<th>Gen</th>
<th>Srcs</th>
<th>Pred?</th>
<th>SrcMod</th>
<th>Src Types</th>
<th>Dst Type</th>
<th>Sat?</th>
<th>CondMod?</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>2C</td>
<td>Call</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
<td>D, UD</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>calla</td>
<td>2B</td>
<td>Call Absolute</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
<td>D, UD</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>cbit</td>
<td>4D</td>
<td>Count Bits Set</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td></td>
<td>UB, UW, UD</td>
<td>UD</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>cmp</td>
<td>10</td>
<td>Compare</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td></td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>cmpn</td>
<td>11</td>
<td>Compare NaN</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td></td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>cont</td>
<td>29</td>
<td>Continue</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>dim</td>
<td>0A</td>
<td>Double Precision Floating-Point Immediate Data Move</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>F</td>
<td>DF</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>dp2</td>
<td>57</td>
<td>Dot Product 2</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>dp3</td>
<td>56</td>
<td>Dot Product 3</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>dp4</td>
<td>54</td>
<td>Dot Product 4</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>dph</td>
<td>55</td>
<td>Dot Product Homogeneous</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>24</td>
<td>Else</td>
<td>Flow Control</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td>25</td>
<td>End If</td>
<td>Flow Control</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Mnem.</td>
<td>Hex Opcode</td>
<td>Name</td>
<td>Group</td>
<td>Gen</td>
<td>Srcs</td>
<td>Pred?</td>
<td>SrcMod</td>
<td>Src Types</td>
<td>Dst Type</td>
<td>Sat?</td>
<td>CondMod?</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>-------------------------------------------</td>
<td>--------------------</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>f16to32</td>
<td>14</td>
<td>Half Precision Float to Single Precision Float</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>f32to16</td>
<td>13</td>
<td>Single Precision Float to Half Precision Float</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>W</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>fbh</td>
<td>4B</td>
<td>Find First Bit from MSB Side</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>D, UD</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>fbl</td>
<td>4C</td>
<td>Find First Bit from LSB Side</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>UD</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>frc</td>
<td>43</td>
<td>Fraction</td>
<td>Parallel Arithmetic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>2A</td>
<td>Halt</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>22</td>
<td>If</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>illegal</td>
<td>00</td>
<td>Illegal</td>
<td>Special</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmpi</td>
<td>20</td>
<td>Jump Indexed</td>
<td>Flow Control</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>D</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>59</td>
<td>Line</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>lrp</td>
<td>5C</td>
<td>Linear Interpolation</td>
<td>Vector Arithmetic</td>
<td>3</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>lzd</td>
<td>4A</td>
<td>Leading Zero Detection</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>D, UD</td>
<td>UD</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mnem.</td>
<td>Hex Opcode</td>
<td>Name</td>
<td>Group</td>
<td>Gen</td>
<td>Srcs</td>
<td>Pred?</td>
<td>SrcMod</td>
<td>Src Types</td>
<td>Dst Type</td>
<td>Sat?</td>
<td>CondMod?</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>-----------------------</td>
<td>---------------------</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>mac</td>
<td>48</td>
<td>Multiply Accumulate</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>mach</td>
<td>49</td>
<td>Multiply Accumulate High</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>mad</td>
<td>5B</td>
<td>Multiply Add</td>
<td>Parallel Arithmetic</td>
<td>3</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>math</td>
<td>38</td>
<td>Extended Math Function</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>N</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>01</td>
<td>Move</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>movi</td>
<td>03</td>
<td>Move Indexed</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>41</td>
<td>Multiply</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>7E</td>
<td>No Operation</td>
<td>Special</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>04</td>
<td>Logic Not</td>
<td>Move and Logic</td>
<td>1</td>
<td>Y</td>
<td>**</td>
<td>Int</td>
<td>Int</td>
<td>N</td>
<td>Equality only</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>06</td>
<td>Logic Or</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>**</td>
<td>Int</td>
<td>Int</td>
<td>N</td>
<td>Equality only</td>
<td></td>
</tr>
<tr>
<td>pln</td>
<td>5A</td>
<td>Plane</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mnem.</td>
<td>Hex Opcode</td>
<td>Name</td>
<td>Group</td>
<td>Gen</td>
<td>Srcs</td>
<td>Pred?</td>
<td>SrcMod</td>
<td>Src Types</td>
<td>Dst Type</td>
<td>Sat?</td>
<td>CondMod?</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>-----------------------</td>
<td>----------------------</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>--------------</td>
<td>----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>ret</td>
<td>2D</td>
<td>Return</td>
<td>Flow Control</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>D, UD</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>rndd</td>
<td>45</td>
<td>Round Down</td>
<td>Parallel Arithmetic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>rnde</td>
<td>46</td>
<td>Round to Nearest or Even</td>
<td>Parallel Arithmetic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>rndu</td>
<td>44</td>
<td>Round Up</td>
<td>Parallel Arithmetic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>rndz</td>
<td>47</td>
<td>Round to Zero</td>
<td>Parallel Arithmetic</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>sad2</td>
<td>50</td>
<td>Sum of Absolute Difference 2</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>B, UB</td>
<td>W, UW</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>sada2</td>
<td>51</td>
<td>Sum of Absolute Difference Accumulate 2</td>
<td>Vector Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>B, UB</td>
<td>W, UW</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>sel</td>
<td>02</td>
<td>Select</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>31</td>
<td>Send Message</td>
<td>Miscellaneous</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>*</td>
<td>*</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>sendc</td>
<td>32</td>
<td>Conditional Send Message</td>
<td>Miscellaneous</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>*</td>
<td>*</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>09</td>
<td>Shift Left</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>Int</td>
<td>Int</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td>08</td>
<td>Shift Right</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>Int</td>
<td>Int</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>subb</td>
<td>4F</td>
<td>Integer Subtraction with Borrow</td>
<td>Parallel Arithmetic</td>
<td>2</td>
<td>Y</td>
<td>N</td>
<td>UD</td>
<td>UD</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td>30</td>
<td>Wait</td>
<td>Miscellaneous</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>UD</td>
<td>UD</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>while</td>
<td>27</td>
<td>While</td>
<td>Flow Control</td>
<td>0</td>
<td>Y</td>
<td>N</td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>07</td>
<td>Logic Xor</td>
<td>Move and Logic</td>
<td>2</td>
<td>Y</td>
<td>**</td>
<td>Int</td>
<td>Int</td>
<td>N</td>
<td>N</td>
<td>Equality only</td>
</tr>
</tbody>
</table>
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.

In the table a cell is gray if it is not applicable because the instruction is not supported for that generation.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>HSW</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>No restrictions.</td>
</tr>
<tr>
<td><code>addc</code></td>
<td>AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.</td>
</tr>
<tr>
<td><code>and</code></td>
<td>Source modifier is not allowed if source is an accumulator.</td>
</tr>
<tr>
<td><code>asr</code></td>
<td>No restrictions.</td>
</tr>
<tr>
<td><code>avg</code></td>
<td>No restrictions.</td>
</tr>
<tr>
<td><code>bfe</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>bfi1</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>bfi2</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>bfrev</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>cbit</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>Accumulator cannot be destination, implicit or explicit.</td>
</tr>
<tr>
<td><code>cmpn</code></td>
<td>Accumulator cannot be destination, implicit or explicit.</td>
</tr>
<tr>
<td><code>csel</code></td>
<td></td>
</tr>
<tr>
<td><code>dim</code></td>
<td>No restrictions.</td>
</tr>
<tr>
<td><code>dp2</code></td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td><code>dp3</code></td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td><code>dp4</code></td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td><code>dph</code></td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td><code>f16to32</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>f32to16</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>fbh</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>fbl</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td><code>frc</code></td>
<td>No restrictions.</td>
</tr>
<tr>
<td><code>line</code></td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td><code>lrp</code></td>
<td>No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.</td>
</tr>
<tr>
<td><code>lzd</code></td>
<td>Accumulator cannot be destination, implicit or explicit.</td>
</tr>
<tr>
<td><code>mac</code></td>
<td>Accumulator is an implicit source and thus cannot be an explicit source operand.</td>
</tr>
<tr>
<td><code>mach</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>mad</code></td>
<td>No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.</td>
</tr>
<tr>
<td><code>math</code></td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td>Instruction</td>
<td>HSW</td>
</tr>
<tr>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>mov</td>
<td>An accumulator can be a source or destination operand but not both.</td>
</tr>
<tr>
<td>movi</td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td>mul</td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td>not</td>
<td>Source modifier is not allowed if source is an accumulator.</td>
</tr>
<tr>
<td>or</td>
<td>Source modifier is not allowed if source is an accumulator.</td>
</tr>
<tr>
<td>pln</td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td>rndd</td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td>rnde</td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td>rndu</td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td>rndz</td>
<td>No accumulator access, implicit or explicit.</td>
</tr>
<tr>
<td>sad2</td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td>sada2</td>
<td>Source operands cannot be accumulators.</td>
</tr>
<tr>
<td>sel</td>
<td>No restrictions.</td>
</tr>
<tr>
<td>shl</td>
<td>Accumulator cannot be destination, implicit or explicit.</td>
</tr>
<tr>
<td>shr</td>
<td>No restrictions.</td>
</tr>
<tr>
<td>subb</td>
<td>AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.</td>
</tr>
<tr>
<td>xor</td>
<td>Source modifier is not allowed if source is an accumulator.</td>
</tr>
</tbody>
</table>
Instruction Set Reference

This chapter describes the functions of 3D Media GPGPU Execution Units, listed in alphabetical order according to assembly language mnemonic.

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.

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;
}
<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
</tr>
<tr>
<td>Addition with Carry</td>
</tr>
<tr>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td>Average</td>
</tr>
<tr>
<td>Bit Field Extract</td>
</tr>
<tr>
<td>Bit Field Insert 1</td>
</tr>
<tr>
<td>Bit Field Insert 2</td>
</tr>
<tr>
<td>Bit Field Reverse</td>
</tr>
<tr>
<td>Branch Converging</td>
</tr>
<tr>
<td>Branch Diverging</td>
</tr>
<tr>
<td>Break</td>
</tr>
<tr>
<td>Call</td>
</tr>
<tr>
<td>Call Absolute</td>
</tr>
<tr>
<td>Compare</td>
</tr>
<tr>
<td>Compare NaN</td>
</tr>
<tr>
<td>Conditional Send Message</td>
</tr>
<tr>
<td>Continue</td>
</tr>
<tr>
<td>Count Bits Set</td>
</tr>
<tr>
<td>Dot Product 2</td>
</tr>
<tr>
<td>Dot Product 3</td>
</tr>
<tr>
<td>Dot Product 4</td>
</tr>
<tr>
<td>Dot Product Homogeneous</td>
</tr>
<tr>
<td>Double Precision Floating Point Immediate Data Move</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>End If</td>
</tr>
<tr>
<td>Extended Math Function</td>
</tr>
<tr>
<td>Find First Bit from LSB Side</td>
</tr>
<tr>
<td>Find First Bit from MSB Side</td>
</tr>
<tr>
<td>Half Precision Float to Single Precision Float</td>
</tr>
<tr>
<td>Halt</td>
</tr>
<tr>
<td>If</td>
</tr>
<tr>
<td>Illegal</td>
</tr>
<tr>
<td>Integer Subtraction with Borrow</td>
</tr>
<tr>
<td>Jump Indexed</td>
</tr>
<tr>
<td>Leading Zero Detection</td>
</tr>
<tr>
<td>Line</td>
</tr>
<tr>
<td>Linear Interpolation</td>
</tr>
<tr>
<td>Logic And</td>
</tr>
</tbody>
</table>

1077
<table>
<thead>
<tr>
<th>Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Not</td>
<td></td>
</tr>
<tr>
<td>Logic Or</td>
<td></td>
</tr>
<tr>
<td>Logic Xor</td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td></td>
</tr>
<tr>
<td>Move Indexed</td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
</tr>
<tr>
<td>Multiply Accumulate</td>
<td></td>
</tr>
<tr>
<td>Multiply Accumulate High</td>
<td></td>
</tr>
<tr>
<td>Multiply Add</td>
<td></td>
</tr>
<tr>
<td>No Operation</td>
<td></td>
</tr>
<tr>
<td>Plane</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td></td>
</tr>
</tbody>
</table>

Round Instructions
- Round Down
- Round to Nearest or Even
- Round to Zero
- Round Up

Select
Send Message
Shift Left
Shift Right
Single Precision Float to Half Precision Float
Sum of Absolute Difference 2
Sum of Absolute Difference Accumulate 2
Wait Notification
While

**EUISA Structures**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddrSubRegNum</td>
</tr>
<tr>
<td>DstRegNum</td>
</tr>
<tr>
<td>DstSubRegNum</td>
</tr>
<tr>
<td>EU_INSTRUCTION_BASIC_ONE_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_BASIC_THREE_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_BASIC_TWO_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_BRANCH_CONDITIONAL</td>
</tr>
<tr>
<td>EU_INSTRUCTION_BRANCH_ONE_SRC</td>
</tr>
</tbody>
</table>
### EUISA Enumerations

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddrMode</td>
</tr>
<tr>
<td>ChanEn</td>
</tr>
<tr>
<td>ChanSel</td>
</tr>
<tr>
<td>CondModifier</td>
</tr>
<tr>
<td>DataType</td>
</tr>
<tr>
<td>DepCtrl</td>
</tr>
<tr>
<td>EU_OPCODE</td>
</tr>
<tr>
<td>ExecSize</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU_INSTRUCTION_BRANCH_TWO_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_COMPACT_TWO_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_CONTROLS</td>
</tr>
<tr>
<td>EU_INSTRUCTION_CONTROLS_A</td>
</tr>
<tr>
<td>EU_INSTRUCTION_CONTROLS_B</td>
</tr>
<tr>
<td>EU_INSTRUCTION_FLAGS</td>
</tr>
<tr>
<td>EU_INSTRUCTION_HEADER</td>
</tr>
<tr>
<td>EU_INSTRUCTION_ILLEGAL</td>
</tr>
<tr>
<td>EU_INSTRUCTION_IMM64_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_MATH</td>
</tr>
<tr>
<td>EU_INSTRUCTION_NOP</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_CONTROLS</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_DST_ALIGN1</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_DST_ALIGN16</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_SEND_MSG</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN1</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_SRC_REG_ALIGN16</td>
</tr>
<tr>
<td>EU_INSTRUCTION_OPERAND_SRC_REG_THREE_SRC</td>
</tr>
<tr>
<td>EU_INSTRUCTION_SEND</td>
</tr>
<tr>
<td>EU_INSTRUCTION_SOURCES_IMM32</td>
</tr>
<tr>
<td>EU_INSTRUCTION_SOURCES_REG</td>
</tr>
<tr>
<td>EU_INSTRUCTION_SOURCES_REG_IMM</td>
</tr>
<tr>
<td>EU_INSTRUCTION_SOURCES_REG_REG</td>
</tr>
<tr>
<td>ExtMsgDescpt</td>
</tr>
<tr>
<td>FunctionControl</td>
</tr>
<tr>
<td>MsgDescpt31</td>
</tr>
<tr>
<td>SrcRegNum</td>
</tr>
<tr>
<td>SrcSubRegNum</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>FC</td>
</tr>
<tr>
<td>HorzStride</td>
</tr>
<tr>
<td>PredCtrl</td>
</tr>
<tr>
<td>QtrCtrl</td>
</tr>
<tr>
<td>RegFile</td>
</tr>
<tr>
<td>RepCtrl</td>
</tr>
<tr>
<td>SFID</td>
</tr>
<tr>
<td>SrcIndex</td>
</tr>
<tr>
<td>SrcMod</td>
</tr>
<tr>
<td>ThreadCtrl</td>
</tr>
<tr>
<td>VertStride</td>
</tr>
<tr>
<td>Width</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Destination Register Type</th>
<th>Default Execution Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UB</td>
<td>B</td>
</tr>
<tr>
<td>UW</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>UD</td>
</tr>
</tbody>
</table>

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<symboll>Expression
```

Notes:

- **.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 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx// r2 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx// r3 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

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
Fully-Expressed Instr

mov(8):f[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.

Assembly Programming Guideline

The following program skeleton illustrates the basic structure of a typical assembly program.

```plaintext
// single line comment
/*          block comment */
<preproc_directive>   // macros, include, etc. Are global – handled by the pre-
<preproc_directive>   // processor
<preproc_directive>   // applies to all code that follows in sequence

// ------------ some kernel
.keras <kernel_name_string>  // [REQUIRED]

// ------ Register requirements -------
.reg_count_total <uint>      // [REQUIRED] a more direct way to specify the parameters
.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>
.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.

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \text{SWZr1r0.0-zx-1} \]

// 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.

\[
\begin{align*}
\text{mul}(8) & \quad r1.0<1> : fr0.xzxz0x1F111F11:v \quad \text{Constant vector of (1 -1 1 1 -1 1 1)} \\
\text{mov (1)} & \quad f0.0 8b'10011001:// \text{Set flag & masked out channels y and z} \\
(f0.0) & \text{mov(8)} r1.0<1> : f 0x000F000F:v \quad \text{Constant vector of (0 0 0 -1 0 0 0 -1)}
\end{align*}
\]

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:

\[ \text{SWZr1r0.0zx-1} \]

// Expected results: \( r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1 \)

GEN:

\[
\begin{align*}
\text{mov (1)} & \quad f0.0 8b'01100110:// \text{Set flag and masked out channels x and w} \\
(f0.0) & \text{sel (8)} r1.0<1> : f r0.yzxy 0x000F000F:v \quad \text{Constant vector of (0 0 0 -1 0 0 0 -1)}
\end{align*}
\]
Destination Mask for DP4 and Destination Dependency Control

The following example demonstrates the use of destination mask mode of floating point dot-product instruction as well as the use of destination dependency control to improve performance (i.e., avoiding unnecessary thread switch due to possible false dependencies).

Consider a generic DirectX 10 Vertex Shader macro of matrix-vector product that is implemented on GEN in the pair of 4-component vector mode. The DirectX 10 equivalent Shader instructions are as the following.

dp4 r5.x r0 r4
dp4 r5.y r1 r4
dp4 r5.z r2 r4
dp4 r5.w r3 r4

With destination dependency control, the GEN instructions are as the following. The first instruction in the sequence checks for the destination dependency, but does not clear the dependency bit. The subsequent two instructions would do neither of them. The last instruction avoids checking the destination dependency, but at completion, it clears the destination scoreboard. It ensures that the content of the destination register is coherent, if any of the following instructions uses the same register as source.

dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr}
dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}
dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}
dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f {NoDDChk}

Just as a comparison, IF GEN DP4 implies reduction at the destination; additional shifted moves are required to achieve the same results. The corresponding codes are as the following. The lower performance due to the additional three move instruction as well as added back-to-back dependencies shows that why we choose to implement the destination channel replication for floating point DP4.

dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.1<1>:f r8.0<1;1,1>:f
dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.2<1>:f r8.0<1;1,1>:f
dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.3<1>:f r8.0<1;1,1>:f
dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f
Null Register as the Destination

Null register can be used as the destination for most of the instructions. Here are some example usages.

- Null as destination for regular ALU instructions: As all ALU instructions can be configured to update the flag registers using the conditional modifiers, it is not necessary to have a destination register if the programmer only cares about the conditionals of the operation. In that case, a null in the destination operand field saves register space as well as one less dependency checking.

- Null as the destination for SEND/STOR instructions: for the send instruction that only send messages out to an external unit and does not require any return data or feedback, a null in the destination register field signifies the case.
  
  o One extension of such case is that even though the operation does not have any return values, a return phase with no payload but simply updating the scoreboard flag for a non-null register can provide a signaling mechanism between the thread and the target external unit. One application of this usage is to allow software to manage the coherency of shared memory resources such like the many caches in the system (particularly, valuable for read/write caches).

Use of LINE Instruction

LINE instruction is specifically designed to speed up floating point vector/matrix computation when a program operates in channel serial.

The following example demonstrates how to use LINE instruction to compute Line Equations for DirectX 10 Pixel Shader. In this example, 2 sets of (Cx#, Cy#, don’t Care, C0#) 4-tuple coefficient vectors are stored in registers R1.

R1: Cx0 Cy0 DC Co0 Cx1 Cy1 DC Co1

8 sets of coordinate 2-D vectors (X, Y) are stored in R2 and R3 in the channel serial mode as

R2: X0 X1 … X7
R3: Y0 Y1 … Y7

The objective is to compute the following two line equations for each set of 2D coordinate and store the results in R4 and R5 as

R4: (X0*Cx0 + Y0*Cy0+Co0) … (X7*Cx0 + Y7*Cy0+Co0)
R5: (X0*Cx1 + Y0*Cy1+Co1) … (X7*Cx1 + Y7*Cy1+Co1)

Example LINE Equations

//------------
// Example compute LINE equation in channel serial scenario
//------------
line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f// does acc = X# * Cx0 + Co0
mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does r4.# = Y# * Cy0 + acc.#
line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f// does acc = X# * Cx0 + Co0
mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f// does r4.# = Y# * Cy0 + acc.#

The next example is to compute homogeneous dot product for OpenGL pixel shader running in Channel Serial. In this example, an original OpenGL PS instruction is like

dph R2.x R0 R1

With register remapping, we can store the input coefficient vector R0 in original format in r0, but 8 sets of input coordinate vectors in channel serial format in r2, r3, r4 and r5, and the destination R2.x component in r6.

r0: Cx0 Cy0 Cz0 Co0 DC DC DC DC
r2: X0 X1 ... X7
r3: Y0 Y1 ... Y7
r4: Z0 Z1 ... Z7
r5: W0 W1 ... W7

The objective is to compute the following DPH equations and store the results in r6 as

R6: (X0*Cx0+Y0*Cy0+Z0*Cz0+Co0) ... (X7*Cx0+Y7*Cy0+Z7*Cz0+Co0)

Example Homogeneous Dot Product in Channel Serial

//---------
// Example compute homogeneous dot product in channel serial scenario
//---------

line (8) acc:f r0<0;1,0>:f r2<0;8,1>:f// does acc = X# * Cx0 + Co0
mac (8) acc:f r0.1<0;1,0>:f r3<0;8,1>:f// does acc.# = Y# * Cy0 + acc.#
mac (8) r6<1>:f r0.2<0;1,0>:f r4<0;8,1>:f// does r6.# = Z# * Cz0 + acc.#

Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of DirectX Shader instructions.

Channel Enables for Extended Math Unit

The following example demonstrates how to use the SEND instruction to get service from the Extended Math unit.

Let's consider COS instruction in DirectX 10 in the following form

`[[(!]p0.{select|any|all})] cos[_sat] dest[mask], [-]src0[_abs][.swizzle]`
For a SIMD4x2 VS implementation with the following register mappings

\[ p0 = \rightarrow f0.0 \]
\[ \text{src0} = \rightarrow r0 \]
\[ \text{dest} = \rightarrow r1 \]

The equivalent GEN instruction is as the following

\[
\begin{array}{c}
\text{SEND (8) } r1\{.mask\}:f m0 \text{ [-]} \text{[abs]}r0\{.swizzle\}:f \text{ MATHBOX|COS|SAT}
\end{array}
\]

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:

\[
\text{MOV (8) } m0\{.mask\}:f \text{ [-]} \text{[abs]}r0.0\{.swizzle\}:f \text{ (NoMask)}
\]

For a SIMD16 PS implementation, the register mappings are as the followings

\[ p0 = \rightarrow f0…f3 \text{ // in order of R, G, B, A} \]
\[ \text{src0} = r0,r1; r2,r3; r4,r5; r6,r7 \]
\[ \text{dest} = r8,r9; r10,r11; r12,r13; r14,r15 \]

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

\[
\begin{array}{c}
\text{send (8) } r8:f \text{ m0 } \text{-(abs)}r2:f \text{ MATHBOX|COS}
\end{array}
\]
\[
\begin{array}{c}
\text{send (8) } r9:f \text{ m1 } \text{-(abs)}r3:f \text{ MATHBOX|COS (SecHalf)}// \text{ use the second half of } 8 \text{ flag bits}
\end{array}
\]
\[
\begin{array}{c}
\text{mov (16) } r10:f \text{ r8:f} \text{ // All destination color channels are same}
\end{array}
\]
\[
\begin{array}{c}
\text{mov (16) } r12:f \text{ r8:f} \text{ // MOV is faster than most MathBox functions}
\end{array}
\]
\[
\begin{array}{c}
\text{mov (16) } r14:f \text{ r8:f} \text{ // These MOVs are compressed instructions}
\end{array}
\]

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).

\[
\begin{array}{c}
(f0.[.any4v,.all4v]) \text{ send (8) } r8:f \text{ m0 } \text{-(abs)}r2:f \text{ MATHBOX|COS}
\end{array}
\]
\[
\begin{array}{c}
(f0.[.any4v,.all4v]) \text{ send (8) } r9:f \text{ m1 } \text{-(abs)}r3:f \text{ MATHBOX|COS (SecHalf)}
\end{array}
\]
(f1[.any4v,.all4v]) mov (16) r10:fr8:f // All destination color channels are same
(f2[.any4v,.all4v]) mov (16) r12:fr8:f // MOV is faster than most MathBox functions
(f3[.any4v,.all4v]) mov (16) r14:fr8:f // These MOVs are compressed instructions

The same instructions work also for predication with select component modifier. We simply replace 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]0[swizzle]:f [-][abs]10[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]0[swizzle]:f [-][abs]1[swizzle]:f
```

1092
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

Notice that with a null as the posted destination register, we are able to transfer the [.mask] over the message channel enables. In many cases for scratch memory assess, a write-with-commit is required, therefore, the posted destination register could be r10.

Now, let's consider the PS case when a scratch register is the destination of an instruction.

\[ p0 => f0 - f4 \]
\[ src0 => r0 - r7 \]
\[ src1 => r8 - r15 \]
\[ dest => s16 - s23 / r16 - r23 \]

When predication is not on (or predication with swizzle control on), we have

\[ \text{add (16) m4:f} [-][\text{abs}]r0/2/4/6\text{BasedOnSwizzle:f} [-][\text{abs}]r8/10/12/14\text{BasedOnSwizzle:f} \]
\[ \text{add (16) m6:f} [-][\text{abs}]r0/2/4/6\text{BasedOnSwizzle:f} [-][\text{abs}]r8/10/12/14\text{BasedOnSwizzle:f} \]
\[ \text{add (16) m8:f} [-][\text{abs}]r0/2/4/6\text{BasedOnSwizzle:f} [-][\text{abs}]r8/10/12/14\text{BasedOnSwizzle:f} \]
\[ \text{add (16) m10:f} [-][\text{abs}]r0/2/4/6\text{BasedOnSwizzle:f} [-][\text{abs}]r8/10/12/14\text{BasedOnSwizzle:f} \]

\[ \text{mov (8) r3:d 0x76543210:v {NoMask}}/\text{ramp function} \]
\[ \text{mul (16) acc0:d r3:d 16 {NoMask}/ramp function} \]
\[ \text{add (8) acc0:d acc0:d 64 {NoMask,SecHalf}/ramp function} \]
\[ \text{add (16) m2:d acc0:d 2*256 {NoMask}}/\text{ramp function} \]

\[ \text{send (16) null } m1 \text{ r3 DATAPORT|RC|WRITE_SIMD16} \]

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.

\[ \text{mov (8) r3:d 0x76543210:v {NoMask}}/\text{ramp function} \]
\[ \text{mul (16) acc0:d r3:d 16 {NoMask}/ramp function} \]
\[ \text{add (8) acc0:d acc0:d 64 {NoMask,SecHalf}/ramp function} \]
\[ \text{add (16) m2:d acc0:d 2*256 {NoMask}}/\text{ramp function} \]

\[ \text{send (16) } r16 \text{ m1 r3 DATAPORT|RC|READ_SIMD16 // read from scratch} \]

// some of the following four instructions may be omitted based on [.mask] field
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) IP4*16 // jumps to HERE_1
add (1) IP0x35 // jumps to HERE_1 (4 lsbs don't-care) <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.fo (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 endifs are required 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
Example Compound Branch Using 'Any' or 'All'

//-------------
// Example: assuming we are doing a channel-serial vector in r0-r3
// We want to know if all components of the vector are > 0x80
//-------------

... cmp.g.f0 (16) null r0 0x80 // r0 > 0x80?
  cmp.g.f1 (16) null r1 0x80 // r1 > 0x80?
  cmp.g.f2 (16) null r2 0x80 // r0 > 0x80?
  cmp.g.f3 (16) null r3 0x80 // r1 > 0x80?
  (f0.all4v) if (16) HERE_1
...

...// code executed only for those channels
...// where per-channel r0,r1,r2,r3 all > 0x80
...

HERE_1:endif
...// and continue...

Looping

Due to GEN's SIMD-16 architecture, it must support the case of up to 16 loops running in parallel. These must be handled as independent loops, each with its own loop-exit condition which could occur after a different number of loop iterations. To account for each channel's progress, a 16b loop-mask LMask is defined with 1b associated to each execution channel. This mask keeps track of which channels remain active inside a loop block.

Basic Do-While Loop

Looping illustrates the most basic loop. Two operations must be accomplished before loop entry. (1) Prior to loop entry, there is some subset of enabled channels as dictated by the code sequence prior. In general, the active status of each channel is indicated in the virtual EMask any point in time. These active channels will become the channels over which the loop is run, and LMask must be initialized with the

1098
EMask value. (2) Since a given loop may be nested within another loop, the previous LMask & CMask must be saved to the LStack for later restoration upon loop completion. The msave instruction performs both the save and update in a single instruction, and thus all loop-blocks should be fronted with a msave LStack LMask and msave LStack CMask operation.

Note that the LMask and CMask share the same mask-stack. Thus, CMask must always be a 1's-subset of the LMask for proper stack operation. This is the case if CMask is updated to LMask each pass through the loop (see Looping) and through the break instruction updating both masks.

Each pass through the loop, a loop terminating operation must be evaluated and stored in a flag register. This condition must be evaluated on a channel-by-channel basis as exemplified:

cmp.z.f0(8) null r2 d3// any operation that updates a flag

The result of this operation sets a bit per channel in the specified flag register, which is then used in the while instruction. As loops are performed, channels may become disabled as their termination condition is met.

While termination is determined on a channel-by-channel basis by the logical AND of corresponding bit positions of AMask, CMask and the specified flag. If the result is 1 the channel remains enabled for the next pass of the loop; if 0 the channel is disabled until loop fall-through. The while instruction causes the LMask to be updated with the latest result of enabled channels. If any channel remains enabled (LMask != ...000b), an additional pass through the loop is made. Once a channel is terminated for the loop operation, it remains terminated until the loop is complete for all channels.

Upon fall through, the while instruction causes the previously saved LMask & CMask to be popped from the LStack, enabling execution on the same subset of channels enabled prior to loop entry (unless a channel had been otherwise terminate inside the loop via halt).

Example Basic Loop Construct

//-----------------------
//Example: Basic do-while loop structure
//-----------------------

... do// save L/CMask & update
BEGIN_LOOP:
    mov (1) CMask LMask(NoMask);// update CMask for this pass
...
<some flag update>

(<>p>)while (8) BEGIN_LOOP// cond. branch
// + restores LMask on fall-through
Do-While Loop with Break

A loop may also be terminated for any channel via the `break` instruction. The `break` instruction causes the corresponding bit positions of enabled channels to be cleared in the LMask. If the updated LMask = ...000b, a branch is made to the specified instruction location. An example is shown below in which the `break` is at the same conditional-nesting level as the terminating `while`. Its primary value may simply be to support a `do...break.. while (true)` –type structure for a more direct 1:1 translation from higher-level source code.

**Example Loop Construct With Non-Nested Break**

```c
//-------
//Example: While-true loop
//-------
#define BrkCode(i,d)(i << 16) + d
do // save L/CMask & update
BEGIN_LOOP:
mov (1) CMask LMask{NoMask} // update CMask for this pass
...
<some flag update>
(<p>)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 Ifs**

```c
//-------
//Example: General Loop Structure w/ break inside Ifs
//-------
```
```c
#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 ...
    ...
    (<p>)break (8) BrkCode(3,HERE_1)// we are 3 levels deep, so
    ...
    endif
    endif
    endif
    ...
    (<p>)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
  ...
  ...
  (<p>) cont (8) ContCode(0,HERE_1)
  ...
HERE_1:
  (<p>) while (8) BEGIN_LOOP// cond. branch
  // + restores C/LMask on fall-through
  ...

Indexed Jump

Example Indexed Jump
//---------
  // Code example shows the use of jmpi to perform a case statement
  // of any number of options in 3 jumps
  //---------
  .default_execution_size 8
  ...
  jmpi r0<0,1,0>// jump relative, based on r0.a.x
  // ------ Jump Table ------
  jmp HERE_0// redirect for case 0
  jmp HERE_1// redirect for case 1
  jmp HERE_2// redirect for case 2
  jmp HERE_3// redirect for case 3
  ...
  HERE_0:// ... case 0 ...

1102
ja	mp DONE
HERE_1:// ... case 1 ...
...
ja	mp DONE
HERE_2:// ... case 2 ...
...
ja	mp DONE
HERE_3:// ... case 3 ...
...
DONE:
...// and continue...