Intel® 965 Express Chipset Family and Intel® G35 Express Chipset Graphics Controller PRM

Programmer's Reference Manual (PRM)

Volume 4: **Subsystem and Cores**

January 2008

Revision 1.0d

Technical queries: ilg@linux.intel.com

www.intellinuxgraphics.org

Contents

1 Introduction 19

1.1 Notations and Conventions 21

1.1.1 Reserved Bits and Software Compatibility 21

1.2 Terminology 21

2 Subsystem Overview 33

2.1 Introduction 33

2.2 Subsystem Topology 33

2.3 Execution Units (EUs) 34

2.4 Thread Dispatching 34

2.5 Shared Functions 35

2.6 Messages 36

2.6.1 Message Register File (MRF) 38

2.6.2 Send Instruction 38

2.6.3 Creating and Sending a Message 39

2.6.4 Message Payload Containing a Header 40

2.6.5 Writebacks 40

2.6.6 Message Delivery Ordering Rules 40

2.6.7 Execution Mask and Messages 41

2.6.8 End-Of-Thread (EOT) Message 41

2.6.9 Performance 42

2.6.10 Message Description Syntax 43

2.6.11 Message Errors 43

3 Debugging 45

3.1 Introduction 45

3.2 The Snapshot Mechanism 46

3.2.1 Debug Trigger Counters 47

3.3 Fixed Function Debug Process 47

3.3.1 Overview 47

3.3.2 Vertex Shader Debug 48

3.3.3 SVG Debug 49

3.3.3.1 SVG_CTL —Debug Control 49

3.3.3.2 SVG_RDATA—Debug Return Data 49

3.3.3.3 SVG_WORK_CTL—Debug Workaround Control 50

3.3.4 Vertex Fetch 50

3.3.4.1 VF_CTL—Debug Control 51

3.3.4.2 VF_STRG_VAL—Debug Snapshot Trigger Value 52

3.3.4.3 VF_STR_VL_OVR —Debug Start Vertex Location Override52

3.3.4.4 VF_VC_OVR —Debug Vertex Count Override 52

3.3.4.5 VF_STR_PSKIP —Debug Starting Primitives Skipped 53

3.3.4.6 VF_MAX_PRIM —Debug Max Primitives 53

3.3.4.7 VF_RDATA —Debug Return Data 53

3.3.5 Vertex Shader 54

3.3.5.1 VS_CTL —Debug Control 55

3.3.5.2 VS_STRG_VAL—Debug Snapshot Trigger Value 55

3.3.5.3 VS_RDATA —Debug Return Data ... 56
3.3.6 Geometry Shader ... 56
3.3.6.1 GS_CTL —Debug Control ... 57
3.3.6.2 GS_STRG_VAL —Debug Snapshot Trigger Value ... 58
3.3.6.3 GS_RDATA —Debug Return Data ... 58
3.3.7 Clipper ... 59
3.3.7.1 CL_CTL—Debug Control ... 60
3.3.7.2 CL_STRG_VAL—Debug Snapshot Trigger Value ... 61
3.3.7.3 CL_RDATA—Debug Return Data ... 61
3.3.8 Strips Fans ... 62
3.3.8.1 SF_CTL —Debug Control ... 63
3.3.8.2 SF_STRG_VAL—Debug Snapshot Trigger Value ... 64
3.3.8.3 SF_MIN_PR_IND—Debug Minimum Primitive Index ... 64
3.3.8.4 SF_MAX_PR_IND—Debug Maximum Primitive Index ... 64
3.3.8.5 SF_CLIP_RMIN– Debug Clip Rectangle Minimum Coordinates ... 65
3.3.8.6 SF_CLIP_RMAX—Debug Clip Rectangle Maximum Coordinates ... 65
3.3.8.7 SF_RDATA—Debug Return Data ... 65
3.3.9 Windower / Intermediate Z ... 66
3.3.9.1 WIZ_CTL—Debug Control ... 68
3.3.9.2 WIZ_STRG_VAL —Debug Snapshot Trigger Value ... 69
3.3.9.3 WIZ_RDATA—Debug Return Data ... 69
3.3.10 Video Front End ... 70
3.3.10.1 VFE_CTL—Debug Control ... 70
3.3.10.2 VFE_STRG_VAL—Debug Snapshot Trigger Value ... 70
3.3.10.3 VFE_RDATA—Debug Return Data ... 71
3.3.11 Thread Spawner ... 71
3.3.11.1 TS_CTL—Debug Control ... 72
3.3.11.2 TS_STRG_0-6VAL—Debug Snapshot Trigger R0.6 Value ... 73
3.3.11.3 TS_STRG_0-7VAL—Debug Snapshot Trigger R0.7 Value ... 73
3.3.11.4 TS_RDATA—Debug Return Data ... 73
3.3.11.5 Parent Thread Recommendations ... 74
3.4 Shared Function Debug ... 75
3.4.1 Thread Dispatcher ... 75
3.4.1.1 TD_CTL—Debug Control ... 75
3.4.1.2 TD_CTL2—Debug Control 2 ... 76
3.4.1.3 TD_VF_VS_EMSK—Debug VF/VS Execution Mask ... 77
3.4.1.4 TD_GS_EMSK—Debug GS Execution Mask ... 77
3.4.1.5 TD_CLIP_EMSK—Debug Clipper Execution Mask ... 77
3.4.1.6 TD_SF_EMSK—Debug SF Execution Mask ... 78
3.4.1.7 TD_WIZ_EMSK — Debug WIZ Execution Mask ... 78
3.4.1.8 TD_0-6_EHTRG_VAL—Debug R0.6 External Halt Trigger Value ... 78
3.4.1.9 TD_0-7_EHTRG_VAL—Debug R0.7 External Halt Trigger Value ... 79
3.4.1.10 TD_0-6_EHTRG_MSK—Debug R0.6 External Halt Trigger Mask ... 79
3.4.1.11 TD_0-7_EHTRG_MSK—Debug R0.7 External Halt Trigger Mask ... 79
3.4.1.12 TD_RDATA—Debug Return Data ... 80
3.4.1.13 TD_TS_EMSK—Debug TS Execution Mask ... 80
3.4.2 Math Unit ... 80
3.4.2.1 MATH_CTL—Math Debug Control ... 80
3.4.2.2 MATH_RDATA—Math Debug Return Data ... 82

3.4.3 Instruction / State Cache.......82
3.4.3.1 ISC_CTL—Instruction / State Debug Control.......82
3.4.4 Instruction L1 Cache.......83
3.4.4.1 ISC_L1CA_CTR—Instruction L1 Cache Debug Control.......83
3.4.4.2 ISC_L1CA_RDATA—Instruction L1 Cache Debug Return Data.......83
3.4.4.3 ISC_L1CA_BP_ADR1—Instruction L1 Cache Breakpoint Address 1 Control.......84
3.4.4.4 ISC_L1CA_BP_ADR2—Instruction L1 Cache Breakpoint Address 2 Control.......84
3.4.4.5 ISC_L1CA_BP_OPC1—Instruction L1 Cache Breakpoint Opcode 1 Control.......85
3.4.4.6 ISC_L1CA_BP_OPC2—Instruction L1 Cache Breakpoint Opcode 2 Control.......85
3.4.5 Message Arbiter.......86
3.4.5.1 MA_DEBUG_1—Message Arbiter Debug Control.......87
3.4.6 Sampler.......88
3.4.6.1 SAMPLER_CTL—Sampler Debug Control.......88
3.4.6.2 SAMPLER_RDATA—Sampler Debug Return Data.......89
3.4.7 Data Port.......89
3.4.7.1 DP_CTL—Data Port Debug Control.......89
3.4.7.2 DP_RDATA—Data Port Debug Return Data.......90
3.4.8 Render Cache.......91
3.4.8.1 RC_CTL—RC Debug Control.......91
3.4.8.2 RC_DEF_CLR—RC Debug Force Default Color.......91
3.4.8.3 RC_RDATA—RC Debug Return Data.......92
3.4.9 Unified Return Buffer (URB).......92
3.4.9.1 URB_CTL—URB Debug Control.......92
3.4.9.2 URB_RDATA—URB Debug Return Data.......93
3.4.10 Thread Spawner (TS).......94
3.5 Attention Signaling from EU to Host.......94
3.5.1 EU_CTL—EU Debug Control.......95
3.5.2 EU_ATT—EU Debug Attention.......95
3.5.3 EU_ATT_DATA—EU Debug Attention Data.......95
3.5.4 EU_ATT_CLR—EU Debug Attention Clear.......96
3.5.5 EU_RDATA—EU Debug Return Data.......96
3.6 Breakpoints.......97
3.6.1 Single Stepping.......97
3.6.2 Modification of Instruction Stream.......98
3.7 Message Errors.......98
3.7.1.1 Error-Types Visible to the Arb.......98
3.7.1.2 Non-pipelined S/F Operation.......99

4 Sampling Engine.......101
4.1 Texture Coordinate Processing.......102
4.1.1 Texture Coordinate Normalization.......102
4.1.2 Texture Coordinate Computation.......102
4.2 Texel Address Generation.......103
4.2.1 Level of Detail Computation (Mipmapping).......104
4.2.1.1 Base Level Of Detail (LOD).......104
4.2.1.2 LOD Bias.......105
4.2.1.3 LOD Pre-Clamping.......105
4.2.1.4 Min/Mag Determination.......105
4.2.1.5 LOD Computation Pseudocode.......106

4.2.2 Inter-Level Filtering Setup 107
4.2.3 Intra-Level Filtering Setup 107
4.2.3.1 MAPFILTER_NEAREST 108
4.2.3.2 MAPFILTER_LINEAR 108
4.2.3.3 MAPFILTER_ANISOTROPIC 109
4.2.3.4 MAPFILTER_MONO 110
4.2.4 Texture Address Control 112
4.2.4.1 TEXCOORDMODE_WRAP Mode 113
4.2.4.2 TEXCOORDMODE_MIRROR Mode 113
4.2.4.3 TEXCOORDMODE_MIRROR_ONCE Mode 113
4.2.4.4 TEXCOORDMODE_CLAMP Mode 114
4.2.4.5 TEXCOORDMODE_CLAMPBORDER Mode 114
4.2.4.6 TEXCOORDMODE_CUBE Mode 114
4.3 Texel Fetch 115
4.3.1 Texel Chroma Keying 115
4.3.1.1 Chroma Key Testing 115
4.3.1.2 Chroma Key Effects 116
4.4 Shadow Prefilter Compare 116
4.5 Texel Filtering 117
4.6 Texel Color Gamma Linearization 117
4.7 State 118
4.7.1 BINDING_TABLE_STATE 118
4.7.2 SURFACE_STATE 118
4.7.2.1 For Most Messages 119
4.7.3 SAMPLER_STATE 142
4.7.3.1 For Most Messages 142
4.7.4 SAMPLER_BORDER_COLOR_STATE 153
4.7.5 3DSTATE_CHROMA_KEY 154
4.7.6 3DSTATE_SAMPLER_PALETTE_LOAD0 156
4.8 Messages 157
4.8.1 Initiating Message 157
4.8.1.1 Message Descriptor 158
4.8.1.2 Message Header 159
4.8.1.3 Payload Parameter Definition 160
4.8.1.4 Message Types 162
4.8.1.5 Parameter Types 165
4.8.1.6 SIMD16 Payload 166
4.8.1.7 SIMD8 Payload 167
4.8.1.8 SIMD4x2 Payload 167
4.8.2 Writeback Message 168
4.8.2.1 SIMD16 168
4.8.2.2 SIMD8 169
4.8.2.3 SIMD4x2 170

5 Data Port 171
5.1 Cache Agents 171
5.1.1 Render Cache 172
5.1.2 Data Cache 172
5.1.3 Sampler Cache 172
5.2 Surfaces 172
5.2.1 Surface State Model 173
5.2.2 Stateless Model 173
5.3 Write Commit 173

5.4 Read/Write Ordering 174
5.5 Accessing Buffers 174
5.6 Accessing Media Surfaces 175
5.6.1 Boundary Behavior 175
5.7 Accessing Render Targets 176
5.7.1 Single Source 176
5.7.2 Dual Source [DevCL-B] 176
5.7.3 Replicate Data 177
5.7.4 Multiple Render Targets (MRT) 177
5.8 Flushing the Render Cache 177
5.9 State 177
5.9.1 BINDING_TABLE_STATE 177
5.9.2 SURFACE_STATE 177
5.10 Messages 178
5.10.1 Global Definitions 178
5.10.1.1 Message Descriptor 179
5.10.1.2 Message Header 180
5.10.1.3 Write Commit Writeback Message 181
5.10.2 OWord Block Read/Write 181
5.10.2.1 Message Descriptor 182
5.10.2.2 Message Payload (Write) 183
5.10.2.3 Writeback Message (Read) 183
5.10.3 OWord Dual Block Read/Write 184
5.10.3.1 Message Descriptor 184
5.10.3.2 Message Payload 185
5.10.3.3 Additional Message Payload (Write) 185
5.10.3.4 Writeback Message (Read) 186
5.10.4 Media Block Read/Write 186
5.10.4.1 Message Descriptor 188
5.10.4.2 Message Header 189
5.10.4.3 Message Payload (Write) 191
5.10.4.4 Writeback Message (Read) 191
5.10.5 DWord Scattered Read/Write 192
5.10.5.1 Message Descriptor 193
5.10.5.2 Message Payload 193
5.10.5.3 Additional Message Payload (Write) 194
5.10.5.4 Writeback Message (Read) 195
5.10.6 Render Target Write 196
5.10.6.1 Subspan/Pixel to Slot Mapping 197
5.10.6.2 Message Descriptor 197
5.10.6.3 Message Header 198
5.10.6.4 Stencil and Antialias Alpha Payload 200
5.10.6.5 Color Payload: SIMD8 Single Source 203
5.10.6.6 Color Payload: SIMD16 Replicated Data 204
5.10.6.7 Color Payload: SIMD8 Dual Source [DevCL-B] 205
5.10.6.8 Depth Payload 206
5.10.6.9 Message Sequencing Summary 207
5.10.7 Flush Render Cache 209
5.10.7.1 Message Descriptor 209
5.10.7.2 Message Payload 209

6 Extended Math 211

6.1 Messages 212
6.1.1 Initiating Message 212

6.1.1.1 Message Descriptor 212
6.1.1.2 Scalar and Vector Mode 213
6.1.1.3 Message Payload 215
6.1.2 Writeback Message 216
6.2 Performance 217
6.3 Function Reference 218
6.3.1 INV 218
6.3.2 LOG 218
6.3.3 EXP 219
6.3.4 SQRT 221
6.3.5 RSQ 222
6.3.6 POW 222
6.3.7 SIN 224
6.3.8 COS 225
6.3.9 SINCOS 226
6.3.10 INT DIV 227

7 Message Gateway 230

7.1 Messages 230
7.1.1 Message Descriptor 231
7.1.2 OpenGateway Message 231
7.1.2.1 Message Payload 232
7.1.2.2 Writeback Message 233
7.1.3 CloseGateway Message 233
7.1.3.1 Message Payload 233
7.1.3.2 Writeback Message 234
7.1.4 ForwardMsg Message 234
7.1.4.1 Message Payload 235
7.1.4.2 Writeback Message to Requester Thread 236
7.1.4.3 Writeback Message to Recipient Thread 236
7.1.5 GetTimeStamp Message 237
7.1.5.1 Message Payload 237
7.1.5.2 Writeback Message to Requester Thread 238

8 Unified Return Buffer (URB) 240

8.1 URB Size 240
8.2 URB Access 240
8.3 State 241
8.4 Messages 241
8.4.1 Execution Mask 241
8.4.2 Message Descriptor 242
8.4.3 URB_WRITE 245
8.4.3.1 URB_WRITE Message Header 245
8.4.3.2 URB_WRITE Message Payload 246
8.4.3.3 Writeback Message for URB Entry Allocate 250

9 Execution Unit ISA 252

9.1 Introduction 252
9.1.1 Objective and Scope 252
9.1.2 Terms and Acronyms 252
9.1.3 Formats and Conventions 255

10 EU Data Types 258

10.1 Fundamental Data Types 258

10.2 Numerical Data Types 259
10.2.1 Unsigned Integers 260
10.2.2 Signed Integers 260
10.2.3 Single Precision Floating-Point Numbers 261
10.2.4 Packed Signed Half-Byte Integer Vector 261
10.2.5 Packed 8-bit Restricted Float Vector 262
10.3 Floating Point Modes 264
10.3.1 IEEE Floating Point Mode 264
10.3.1.1 Partial Listing of Honored IEEE-754 Rules 264
10.3.1.2 Complete Listing of Deviations or Additional Requirements vs. IEEE-754 265
10.3.1.3 Comparison of Floating Point Numbers 266
10.3.1.4 Min/Max of Floating Point Numbers 268
10.3.2 Alternative Floating Point Mode 272
10.4 Type Conversion 274
10.4.1 Float to Integer 274
10.4.2 Integer to Integer with Same or Higher Precision 274
10.4.3 Integer to Integer with Lower Precision 275
10.4.4 Integer to Float 275

11 Execution Environment 276
11.1 Overview 276
11.2 Primary Usage Models 277
11.2.1 AOS and SOA Data Structures 277
11.2.2 SIMD4 Mode of Operation 278
11.2.3 SIMD4x2 Mode of Operation 279
11.2.4 SIMD16 Mode of Operation 281
11.2.5 SIMD8 Mode of Operation 282
11.3 Registers and Register Regions 283
11.3.1 Register Files 283
11.3.2 GRF Registers 283
11.3.3 MRF Registers 284
11.3.4 ARF Registers 286
11.3.4.1 Overview 286
11.3.4.2 Access Granularity 287
11.3.4.3 Null Register 287
11.3.4.4 Address Register 288
11.3.4.5 Accumulator Registers 291
11.3.4.6 Flag Register 294
11.3.4.7 Mask Registers 295
11.3.4.8 Mask Stack Register 297
11.3.4.9 Mask Stack Depth Register 299
11.3.4.10 State Registers 302
11.3.4.11 Control Register 304
11.3.4.12 Notification Registers 312
11.3.4.13 IP Register 314
11.3.5 Immediate 315
11.3.6 Region Parameters 316
11.3.7 Region Addressing Modes 321
11.3.7.1 Direct Register Addressing 322
11.3.7.2 Register-indirect Register Addressing with a 1x1 Index Region 323
11.3.7.3 Register-indirect Register Addressing with a Vx1 Index Region 324

11.3.7.4 Register-indirect Register Addressing with a VxH Index Region 325
11.3.8 Access Modes 326
11.3.9 Execution Data Type 327
11.3.10 Register Region Restrictions 327
11.3.10.1 Examples 330
11.3.10.2 Different Raw Moves 332
11.3.11 Destination Operand Description 332
11.3.11.1 Destination Region Parameters 332
11.4 SIMD Execution Control 333
11.4.1 Predication 333
11.4.2 No Predication 334
11.4.3 Predication with Horizontal Combination 335
11.4.4 Predication with Vertical Combination 336
11.5 Instruction Compression 337
11.5.1 Motivation and Expected Usage 337
11.5.2 Hardware Behavior 338
11.5.3 Rules and Restrictions 340
11.5.4 Usage Examples 341
11.6 End of Thread 344
11.7 Creating Conditional Flags 345
11.8 Destination Hazard 348
11.9 Non-present Operands 348
11.10 Instruction Prefetch 349
12 Exceptions 350
12.1 Introduction 350
12.2 Exception-Related Architectural Registers 350
12.3 System Routine 351
12.3.1 General Flow of the System Routine 351
12.3.2 Invoking the System Routine 352
12.3.3 Returning to the Application Thread 353
12.3.4 System-IP (SIP) 353
12.3.5 System Routine Register Space 353
12.3.6 System-Scratch Memory Space 354
12.3.7 Conditional Instructions Within System Routines 355
12.3.8 Messages in System Routines 355
12.3.9 Use of ‘NoDDClr’ 355
12.4 Exception Descriptions 357
12.4.1 ‘Illegal’ opcode 357
12.4.2 Undefined opcode 357
12.4.3 MaskStack Overflow / Underflow 357
12.4.4 Software Exception 358
12.4.5 Breakpoint 358
12.4.6 External Halt 358
12.5 Events Which Do Not Generate Exceptions 359
12.6 System Handler Example 360
13 Instruction Set Summary 364
13.1 Instruction Set Characteristics 364
13.1.1 SIMD Instructions and SIMD Width 364
13.1.2 Instruction Operands and Register Regions 364
13.1.3 Instruction Execution 365

13.2 Instruction Machine Formats 366
13.2.1 Common Instruction Fields 368
13.2.2 Instruction Operation Doubleword (DW0) 374
13.2.3 Instruction Destination Doubleword (DW1) 379
13.2.4 Instruction Source-0 Doubleword (DW2) 383
13.2.5 Instruction Source-1 Doubleword (DW3) 387
13.3 Opcode Encoding 393
13.3.1 Move and Logic Instructions 393
13.3.2 Flow Control Instructions 395
13.3.3 Miscellaneous Instructions 396
13.3.4 Parallel Arithmetic Instructions 397
13.3.5 Vector Arithmetic Instructions 398
13.3.6 Special Instructions 399
13.4 Native Instruction BNF 400
13.4.1 Instruction Groups 400
13.4.2 Destination Register 401
13.4.3 Source Register 402
13.4.4 Address Registers 403
13.4.5 Register Files and Register Numbers 403
13.4.6 Relative Location and Stack Control 404
13.4.7 Regions 404
13.4.8 Types 405
13.4.9 Write Mask 405
13.4.10 Swizzle Control 405
13.4.11 Immediate Values 405
13.4.12 Predication and Modifiers 406
13.4.13 Instruction Options 407
13.5 Deprecated Features 408
13.5.1 Defeatured Instructions 408
13.5.2 Others 408

14 Instruction Set Reference 410
14.1 Conventions 410
14.1.1 Pseudo Code Format 410
14.1.2 General Macros and Definitions 410
14.1.3 Mask Stack Operations 411
14.2 Instruction Description 414
14.2.1 add – Addition 415
14.2.2 and – Logical And 417
14.2.3 asr – Arithmetic Shift Right 419
14.2.4 avg – Average 421
14.2.5 break – Break 422
14.2.6 cmp – Compare 425
14.2.7 cmpn – Compare NaN 427
14.2.8 cont – Continue 429
14.2.9 do – Do 431
14.2.10 dp2 – Dot Product 2 432
14.2.11 dp3 – Dot Product 3 434
14.2.12 dp4 – Dot Product 4 436
14.2.13 dph –Dot Product Homogeneous 438
14.2.14 else – Else 440
14.2.15 endif – End-If 442
14.2.16 frc – Fraction 444
14.2.17 halt – Halt 446

14.2.18 if – If 448
14.2.19 iff – Fast-If 450
14.2.20 jmpi – Jump Indexed 452
14.2.21 line – Line 454
14.2.22 lzd – Leading Zero Detection 456
14.2.23 mac – Multiply Accumulate 458
14.2.24 mach – Multiply Accumulate High 459
14.2.25 mov – Move 461
14.2.26 mrest – Mask Restore 462
14.2.27 msave – Mask Save 463
14.2.28 mul – Multiply 464
14.2.29 nop – No Operation 467
14.2.30 not – Logic Not 468
14.2.31 or – Logic Or 470
14.2.32 pop – Mask Stack Pop 472
14.2.33 push – Mask Stack Push 473
14.2.34 rndd – Round Down 474
14.2.35 rndu – Round Up 476
14.2.36 rnde – Round to Even 478
14.2.37 rndz – Round to Zero 480
14.2.38 sad2 – Sum of Absolute Difference 2 482
14.2.39 sada2 – Sum of Absolute Difference Accumulate 2 484
14.2.40 sel – Select 486
14.2.41 send – Send Message 488
14.2.42 shl – Shift Left 496
14.2.43 shr – Shift Right 497
14.2.44 wait – Wait Notification 498
14.2.45 while – While 500
14.2.46 xor – Logic Xor 502

15 EU Programming Guide 504

15.1 Assembler Pragmas 504
15.1.1 Declarations 504
15.1.2 Defaults and Defines 505
15.1.3 Example Pragma Usages 506
15.1.4 Assembly Programming Guideline 507
15.2 Usage Examples 509
15.2.1 Vector Immediate 509
15.2.1.1 Supporting Pixel Shader Indexing 509
15.2.1.2 Supporting OpenGL Vertex Shader Instruction SWZ 510
15.2.2 Destination Mask for DP4 and Destination Dependency Control 511
15.2.3 Null Register as the Destination 512
15.2.4 Use of LINE Instruction 513
15.2.5 Mask for SEND Instruction 515
15.2.5.1 Channel Enables for Extended Math Unit 515
15.2.5.2 Channel Enables for Scratch Memory 517
15.2.6 Flow Control Instructions 520
15.2.7 Execution Masking 521
15.2.7.1 Branching 521
15.2.7.2 Fast-If 521
15.2.7.3 Cascade Branching 522
15.2.7.4 Compound Branches 522
15.2.7.5 Looping 523
15.2.7.6 Indexed Jump 526

Figures

Figure 2-1. Data Flow Associated With Messages 37
Figure 4-1. Normalized vs. Unnormalized Texture Coordinates........................... 102
Figure 4-2. Cube Map Coordinate Computation Example 103
Figure 4-3. Bilinear Filter Sampling... 109
Figure 4-4. Sampling Using MAPFILTER_MONO.. 111
Figure 4-5. Texture Wrap vs. Mirror Addressing Mode...................................... 113
Figure 4-6. Texture Clamp Mode 114
Figure 10-1. Fundamental data types... 258
Figure 10-2. Integer numerical data types 259
Figure 10-3. Floating point numerical data types 259
Figure 10-4. Converting a Packed Half-byte Vector to a 128-bit Signed Integer Vector261
Figure 10-5. Conversion from a Restricted 8-bit Float to a Single-Precision Float ... 262
Figure 10-6. Converting a Packed Restricted Float Vector to a 128-bit Float Vector 264
Figure 11-1. AOS and SOA data structures 277
Figure 11-2. A SIMD4 Example 279
Figure 11-3. SIMD4x2 Examples with Different Emasks 280
Figure 11-4. A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows 280
Figure 11-5. A SIMD16 Example... 281
Figure 11-6. Another SIMD16 Example with an AOS Shared Constant.................. 282
Figure 11-7. Format of the Mask Stack Register 299
Figure 11-8. Format of the Mask Stack Depth Register 301
Figure 11-9. Conventional SIMD Instruction Sequence..................................... 317
Figure 11-10. GEN4 SIMD Instruction Sequence for the Same Program................ 318
Figure 11-11. An example of a register region (*r4.1<16;8,2>:w*) with 16 elements 320
Figure 11-12. A 16-element register region with interleaved rows (*r5.0<1;8,2>:w*) 320
Figure 11-13. A region description example in direct register addressing.............. 322
Figure 11-14. A region description example in direct register addressing with <src0> as a vector of replicated scalars 323
Figure 11-15. An example illustrating register-indirect register addressing mode with a 1x1 index region.. 324
Figure 11-16. An example illustrating register-indirect-register addressing mode with a Vx1 index region (Src0) 325
Figure 11-17. An example illustrating register-indirect register addressing mode with a VxH index region (Src0).. 326
Figure 11-18. Generation of predication mask 334
Figure 11-19. Direct addressed vector operands in a compressed instruction 341
Figure 11-20. A packed-word operand in a dword SIMD16 compressed instruction . 342
Figure 11-21. Dword and word (or byte) scalar source in a compressed instruction. 343
Figure 11-22. Indirect-addressed source/destination operand in a compressed instruction 343
Figure 15-1. Pixel Shader example using vector immediate 510

Tables

Table 1-1. Supported Chipsets 19
Table 2-1. Error Cases 44
Table 8-1. URB_TRANSPOSE Payload 249
Table 8-2.URB_TRANSPOSE URB Destination Layout 250
Table 10-1. Formats and ranges of numerical data types 260
Table 10-2. Example of restricted 8-bit float numbers 263
Table 10-3. Results of "Greater-Than" Comparison – CMP.G 266
Table 10-4. Results of "Less-Than" Comparison – CMP.L 267
Table 10-5. Results of "Equal-To" Comparison – CMP.E........ 267
Table 10-6. Results of "Not-Equal-To" Comparison – CMP.NE 267
Table 10-7. Results of "Less-Than Or Equal-To" Comparison – CMP.LE 268
Table 10-8. Results of "Greater-Than or Equal-To" Comparison – CMP.GE 268
Table 10-9. Results of "Greater-Than" Comparison-NaN – CMPN.G 269
Table 10-10. Results of "Less-Than" Comparison-NaN – CMPN.L 270
Table 10-11. Results of "Equal-To" Comparison-NaN – CMPN.E 270
Table 10-12. Results of "Not-Equal-To" Comparison-NaN – CMPN.NE 270
Table 10-13. Results of "Less-Than Or Equal-To" Comparison-NaN – CMPN.LE 271
Table 10-14. Results of "Greater-Than or Equal-To" Comparison-NaN – CMPN.GE.. 271
Table 10-15. Supported Legacy Float Mode and Impacted Units 272
Table 10-16. Dismissed legacy behaviors........ 273
Table 11-1. Summary of GRF Registers 284
Table 11-2. GRF Registers Available in Device Hardware 284
Table 11-3. Summary of MRF Registers 285
Table 11-4. MRF Registers Available in Device Hardware........ 285
Table 11-5. Summary of Architecture Registers 286
Table 11-6. Register and Subregister Numbers for Address Register 288
Table 11-7. Address Register Fields 289
Table 11-8. Register and Subregister Numbers for Accumulate Register 291
Table 11-9. Accumulator Channel Precision 293
Table 11-10. Register and Subregister Numbers for Flag Register 295
Table 11-11. Flag Register Fields........ 295
Table 11-12. Register and Subregister Numbers for Mask Register 296
Table 11-13. Mask Register Fields 296
Table 11-14. Register and Subregister Numbers for Mask Stack Register........ 298
Table 11-15. Mask Stack Register Fields 298
Table 11-16. Overflow/Underflow Exception Trigger Value........ 300
Table 11-17. Register and Subregister Numbers for Mask Stack Depth Register 301
Table 11-18. Mask Stack Depth Register Fields........ 301
Table 11-19. Register and Subregister Numbers for State Register 302
Table 11-20. State Register Fields 303
Table 11-21. Register and Subregister Numbers for Control Register 305
Table 11-22. Control Register Fields 305
Table 11-23. Register and Subregister Numbers for Notification Register........ 312
Table 11-24. Fields of Notification Register n0 313
Table 11-25. Fields of Notification Register n1 313
Table 11-26. Format of the Notification Register 313
Table 11-27. Register and Subregister Numbers for IP Register 314
Table 11-28. IP Register Fields........ 314

Table 11-29. Execution size in device hardware 330
Table 11-30. Indirect source addressing support available in device hardware........ 330
Table 13-1. GRF instruction execution parameters in device hardware 364
Table 13-2. GEN4 Instruction Format 367
Table 13-3. Definitions of Common Instruction Fields 368
Table 13-4. Definitions of Fields in Operation Doubleword (DW0) 374
Table 13-5. Instruction Destination Doubleword 379
Table 13-6. Destination Register Region in Direct + Align16 mode 381
Table 13-7. Destination Register Region in Direct+Align1 mode 381
Table 13-8. Destination Register Region in Indirect+Align16 mode 382
Table 13-9. Destination Register Region in Indirect+Align1 mode........................ 382
Table 13-10. Instruction Source-0 Doubleword in Direct+Align16 mode 383
Table 13-11. Instruction Source-0 Doubleword in Direct+Align1 mode 384
Table 13-12. Instruction Source-0 Doubleword in Indirect+Align16 mode............. 385
Table 13-13. Instruction Source-0 Doubleword in Indirect+Align1 mode 386
Table 13-14. Instruction Source-1 Doubleword in Direct + Align16 mode 387
Table 13-15. Instruction Source-1 Doubleword in Direct + Align1 mode 388
Table 13-16. Instruction Source-1 Doubleword in Indirect+Align16 mode............. 389
Table 13-17. Instruction Source-1 Doubleword in Indirect+Align1 mode 390
Table 13-18. GEN4 Compacted Instruction Format 391
Table 13-19. Definitions of Fields in the Compact Instruction.............................. 391
Table 13-20. Move and Logic Instructions 394
Table 13-21. Flow Control Instructions 395
Table 13-22. Miscellaneous Instructions 396
Table 13-23. Parallel Arithmetic Instructions 397
Table 13-24. Vector Arithmetic Instructions 398
Table 13-25. Special Instructions 399
Table 14-1. Floating point addition of A (column) and B (row) in IEEE mode.......... 416
Table 14-2. Floating point addition of A (column) and B (row) in ALT mode 416
Table 14-3. Floating point fraction computation in IEEE mode............................. 445
Table 14-4. Floating point fraction computation in ALT mode.............................. 445
Table 14-5. Floating point multiplication of A (column) and B (row) in IEEE mode .. 465
Table 14-6. Floating point multiplication of A (column) and B (row) in ALT mode.... 465
Table 14-7. Floating point round-down in IEEE mode.. 475
Table 14-8. Floating point round-down in ALT mode.. 475
Table 14-9. Floating point round-up in IEEE mode 477
Table 14-10. Floating point round-up in ALT mode 477
Table 14-11. Floating point round-to-even in IEEE mode.................................... 479
Table 14-12. Floating point round-to-even in ALT mode..................................... 479
Table 14-13. Floating point round-to-zero in IEEE mode 481
Table 14-14. Floating point round-to-zero in ALT mode...................................... 481
Table 14-15. Message Descriptor Definition 492
Table 14-16. Sideband Signals Associated with Each Message Sent to the Shared Function 493

Revision History

Document Number	Revision Number	Description	Revision Date
4	1.0d	Initial release.	January 2008

§§

1 Introduction

This Programmer's Reference Manual (PRM) describes the architectural behavior and programming environment of the Intel® 965 Express Chipset family and Intel® G35 Express Chipset GMCH graphics devices (see Table 1-1). The GMCH's Graphics Controller (GC) contains an extensive set of registers and instructions for configuration, 2D, 3D, and Video systems. The PRM describes the register, instruction, and memory interfaces and the device behaviors as controlled and observed through those interfaces. The PRM also describes the registers and instructions and provides detailed bit/field descriptions.

Note: The term "Gen4" is used throughout the PRM to refer to the Generation 4 family of graphics devices. The devices listed in Table 1-1 are Gen4 devices.

Table 1-1. Supported Chipsets

Chipset Family Name	Device Name	Device Tag
Intel® Q965 Chipset Intel® Q963 Chipset Intel® G965 Chipset	82Q965 GMCH 82Q963 GMCH 82G965 GMCH	[**DevBW**]
Intel® G35 Chipset	82G35 GMCH	[**DevBW-E**]
Intel® GM965 Chipset Intel® GME965 Chipset	GM965 GMCH GME965 GMCH	[**DevCL**]

NOTES:

1. Unless otherwise specified, the information in this document applies to all of the devices mentioned in Table 1-1. For Information that does not apply to all devices, the Device Tag is used.
2. Throughout the PRM, references to "All" in a project field refters to all devices in Table 1-1.
3. Throughout the PRM, references to [DevBW] apply to both [DevBW] and [DevBW-E]. [DevBW-E] is referenced specifically for information that is [DevBW-E] only.
4. Stepping info is sometimes appended to the device tag (e.g., [DevBW-C]). Information without any device tagging is applicable to all devices/steppings.

The PRM is intended for hardware, software, and firmware designers who seek to implement or use the graphic functions of the 965 Express Chipset family and G35 Express Chipset. Familiarity with 2D and 3D graphics programming is assumed.

The Programmer's Reference Manual is organized into four volumes:

- **PRM, Volume 1: Graphics Core**
  Volume 1 covers the overall Graphics Processing Unit (GPU) without much detail on 3D, Media, or the core subsystem. Topics include the command streamer, context switching, and memory access (including tiling). The Memory Data Formats can also be found in this volume.

  The volume also contains a chapter on the Graphics Processing Engine (GPE). The GPE is a collective term for 3D, Media, the subsystem, and the parts of the memory interface that are used by these units. Display, blitter and their memory interfaces are *not* included in the GPE.
- ***PRM, Volume 2; 3D/Media***
  Volume 2 covers the 3D and Media pipelines in detail. This volume is where details for all of the "fixed functions" are covered, including commands processed by the pipelines, fixed-function state structures, and a definition of the inputs (payloads) and outputs of the threads spawned by these units.

  This volume also covers the single Media Fixed Function, VLD. It describes how to initiate generic threads using the thread spawner (TS). It is generic threads which will be used for doing the majority of media functions. Programmable kernels will handle the algorithms for media functions such IDCT, Motion Compensation, and even Motion Estimation (used for encoding MPEG streams).
- ***PRM, Volume 3: Display Registers***
  Volume 3 describes the control registers for the display. The overlay registers and VGA registers are also cover in this volume.
- ***PRM, Volume 4: Subsystem and Cores***
  Volume 4 describes the GMCH programmable cores, or EUs, and the "shared functions", which are shared by more than one EU and perform functions such as I/O and complex math functions.

  The shared functions consist of the sampler, extended math unit, data port (the interface to memory for 3D and media), Unified Return Buffer (URB), and the Message Gateway which is used by EU threads to signal each other. The EUs use messages to send data to and receive data from the subsystem; the messages are described along with the shared functions, although the generic message send EU instruction is described with the rest of the instructions in the Instruction Set Architecture (ISA) chapters.

  This latter part of this volume describes the GMCH core, or EU, and the associated instructions that are used to program it. The instruction descriptions make up what is referred to as an Instruction Set Architecture, or ISA. The ISA describes all of the instructions that the GMCH core can execute, along with the registers that are used to store local data.

Note: The chipset PCI Configuration registers are not part of this PRM.

1.1 Notations and Conventions

1.1.1 Reserved Bits and Software Compatibility

In many register, instruction and memory layout descriptions, certain bits are marked as “Reserved”. When bits are marked as reserved, it is essential for compatibility with future devices that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

Do not depend on the states of any reserved bits when testing values of registers that contain such bits. Mask out the reserved bits before testing. Do not depend on the states of any reserved bits when storing to instruction or to a register.

When loading a register or formatting an instruction, always load the reserved bits with the values indicated in the documentation, if any, or reload them with the values previously read from the register.

1.2 Terminology

Term	Abbr.	Definition
3D Pipeline	—	One of the two pipelines supported in the GPE. The 3D pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See *3D Pipeline*.
Application IP	AIP	Application Instruction Pointer. This is part of the control registers for exception handling for a thread. Upon an exception, hardware moves the current IP into this register and then jumps to SIP.
Architectural Register File	ARF	A collection of architecturally visible registers for a thread such as address registers, accumulator, flags, notification registers, IP, null, etc. ARF should not be mistaken as just the address registers.
Array of Cores	—	Refers to a group of Gen4 EUs, which are physically organized in two or more rows. The fact that the EUs are arranged in an array is (to a great extent) transparent to CPU software or EU kernels.
Binding Table	—	Memory-resident list of pointers to surface state blocks (also in memory).
Binding Table Pointer	BTP	Pointer to a binding table, specified as an offset from the Surface State Base Address register.
Bypass Mode	—	Mode where a given fixed function unit is disabled and forwards data down the pipeline unchanged. Not supported by all FF units.
Byte	B	A numerical data type of 8 bits, B represents a signed byte integer.

Term	Abbr.	Definition
Child Thread	—	A branch-node or a leaf-node thread that is created by another thread. It is a kind of thread associated with the media fixed function pipeline. A child thread is originated from a thread (the parent) executing on an EU and forwarded to the Thread Dispatcher by the TS unit. A child thread may or may not have child threads depending on whether it is a branch-node or a leaf-node thread. All pre-allocated resources such as URB and scratch memory for a child thread are managed by its parent thread.
Clip Space	—	A 4-dimensional coordinate system within which a clipping frustum is defined. Object positions are projected from Clip Space to NDC space via "perspecitive divide" by the W coordinate, and then viewport mapped into Screen Space
Clipper	—	3D fixed function unit that removes invisible portions of the drawing sequence by discarding (culling) primitives or by "replacing" primitives with one or more primitives that replicate only the visible portion of the original primitive.
Color Calculator	CC	Part of the Data Port shared function, the color calculator performs fixed-function pixel operations (e.g., blending) prior to writing a result pixel into the render cache.
Command	—	Directive fetched from a ring buffer in memory by the Command Streamer and routed down a pipeline. Should not be confused with instructions which are fetched by the instruction cache subsystem and executed on an EU.
Command Streamer	CS or CSI	Functional unit of the Graphics Processing Engine that fetches commands, parses them and routes them to the appropriate pipeline.
Constant URB Entry	CURBE	A UE that contains "constant" data for use by various stages of the pipeline.
Control Register	CR	The read-write registers are used for thread mode control and exception handling for a thread.
Data Port	DP	Shared function unit that performs a majority of the memory access types on behalf of Gen4 programs. The Data Port contains the render cache and the constant cache and performs all memory accesses requested by Gen4 programs except those performed by the Sampler. See DataPort.
Degenerate Object	—	Object that is invisible due to coincident vertices or because does not intersect any sample points (usually due to being tiny or a very thin sliver).
Destination	—	Describes an output or write operand.
Destination Size	—	The number of data elements in the destination of a Gen4 SIMD instruction.

Term	Abbr.	Definition
Destination Width	—	The size of each of (possibly) many elements of the destination of a Gen4 SIMD instruction.
Double Quad word (DQword)	DQ	A fundamental data type, DQ represents 16 bytes.
Double word (DWord)	D or DW	A fundamental data type, D or DW represents 4 bytes.
Drawing Rectangle	—	A screen-space rectangle within which 3D primitives are rendered. An objects screen-space positions are relative to the Drawing Rectangle origin. See *Strips and Fans*.
End of Block	EOB	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.
End Of Thread	EOT	a message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set in order to properly terminate.
Exception	—	Type of (normally rare) interruption to EU execution of a thread's instructions. An exception occurrence causes the EU thread to begin executing the System Routine which is designed to handle exceptions.
Execution Channel	—	Gen4 EU instructions typically operate on multiple data values in parallel (i.e., in "SIMD" fashion). The data is processed in parallel "execution channels" (e.g., a SIMD8 instruction uses 8 execution channels to perform 8 operations in parallel).
Execution Size	ExecSize	Execution Size indicates the number of data elements processed by a GEN4 SIMD instruction. It is one of the GEN4 instruction fields and can be changed per instruction.
Execution Unit	EU	Execution Unit. An EU is a multi-threaded processor within the GEN4 multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a GEN4 Core.
Execution Unit Identifier	EUID	The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU a thread is located. A thread can be uniquely identified by the EUID and TID.
Execution Width	ExecWidth	The width of each of several data elements that may be processed by a single Gen4 SIMD instruction.
Extended Math Unit	EM	A Shared Function that performs more complex math operations on behalf of several EUs.
FF Unit	—	A Fixed-Function Unit is the hardware component of a 3D Pipeline Stage. A FF Unit typically has a unique FF ID associated with it.

Term	Abbr.	Definition
Fixed Function	FF	Function of the pipeline that is performed by dedicated (vs. programmable) hardware.
Fixed Function ID	FFID	Unique identifier for a fixed function unit.
FLT_MAX	fmax	The magnitude of the maximum representable single precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's.
Gateway	GW	See Message Gateway.
GEN4 Core	—	Alternative name for an EU in the GEN4 multi-processor system.
General Register File	GRF	Large read/write register file shared by all the EUs for operand sources and destinations. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.
General State Base Address	—	The Graphics Address of a block of memory-resident "state data", which includes state blocks, scratch space, constant buffers and kernel programs. The contents of this memory block are referenced via offsets from the contents of the General State Base Address register. See *Graphics Processing Engine*.
Geometry Shader	GS	Fixed-function unit between the vertex shader and the clipper that (if enabled) dispatches "geometry shader" threads on its input primitives. Application-supplied geometry shaders normally expand each input primitive into several output primitives in order to perform 3D modeling algorithms such as fur/fins. See *Geometry Shader*.
Graphics Address	—	The GPE virtual address of some memory-resident object. This virtual address gets mapped by a GTT or PGTT to a physical memory address. Note that many memory-resident objects are referenced not with Graphics Addresses, but instead with offsets from a "base address register".
Graphics Processing Engine	GPE	Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer.
Guardband	GB	Region that may be clipped against to make sure objects do not exceed the limitations of the renderer's coordinate space.
Horizontal Stride	HorzStride	The distance in element-sized units between adjacent elements of a Gen4 region-based GRF access.
Immediate floating point vector	VF	A numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.

Term	Abbr.	Definition
Immediate integer vector	V	A numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4-bit each. The 4-bit integer element is in 2's complement form. It may be used to specify the type of an immediate operand in an instruction.
Index Buffer	IB	Buffer in memory containing vertex indices.
Instance	—	In the context of the VF unit, an instance is one of a sequence of sets of similar primitive data. Each set has identical vertex data but may have unique instance data that differentiates it from other sets in the sequence.
Instruction	—	Data in memory directing an EU operation. Instructions are fetched from memory, stored in a cache and executed on one or more Gen4 cores. Not to be confused with commands which are fetched and parsed by the command streamer and dispatched down the 3D or Media pipeline.
Instruction Pointer	IP	The address (really an offset) of the instruction currently being fetched by an EU. Each EU has its own IP.
Instruction Set Architecture	ISA	The GEN4 ISA describes the instructions supported by a GEN4 EU.
Instruction State Cache	ISC	On-chip memory that holds recently-used instructions and state variable values.
Interface Descriptor	—	Media analog of a State Descriptor.
Intermediate Z	IZ	Completion of the Z (depth) test at the front end of the Windower/Masker unit when certain conditions are met (no alpha, no pixel-shader computed Z values, etc.)
Inverse Discrete Cosine Transform	IDCT	the stage in the video decoding pipe between IQ and MC
Inverse Quantization	IQ	A stage in the video decoding pipe between IS and IDCT.
Inverse Scan	IS	A stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for MPEG-2.
Jitter	—	Just-in-time compiler.
Kernel	—	A sequence of Gen4 instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to Gen4 instructions.
Least Significant Bit	LSB	Least Significant Bit
MathBox	—	See Extended Math Unit
Media	—	Term for operations such as video decode and encode that are normally performed by the Media pipeline.

Term	Abbr.	Definition
Media Pipeline	—	Fixed function stages dedicated to media and "generic" processing, sometimes referred to as the generic pipeline.
Message	—	Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of GEN4 architecture.
Message Gateway	—	Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline.
Message Register File	MRF	Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction.
Most Significant Bit	MSB	Most Significant Bit
Motion Compensation	MC	Part of the video decoding pipe.
Motion Picture Expert Group	MPEG	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.
Motion Vector Field Selection	MVFS	A four-bit field selecting reference fields for the motion vectors of the current macroblock.
Multi Render Targets	MRT	Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state.
Normalized Device Coordinates	NDC	Clip Space Coordinates that have been divided by the Clip Space "W" component.
Object	—	A single triangle, line or point.
Parent Thread	—	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Pipeline Stage	—	A abstracted element of the 3D pipeline, providing functions performed by a combination of the corresponding hardware FF unit and the threads spawned by that FF unit.
Pipelined State Pointers	PSP	Pointers to state blocks in memory that are passed down the pipeline.
Pixel Shader	PS	Shader that is supplied by the application, translated by the jitter and is dispatched to the EU by the Windower (conceptually) once per pixel.
Point	—	A drawing object characterized only by position coordinates and width.
Primitive	—	Synonym for object: triangle, rectangle, line or point.

Term	Abbr.	Definition
Primitive Topology	—	A composite primitive such as a triangle strip, or line list. Also includes the objects triangle, line and point as degenerate cases.
Provoking Vertex	—	The vertex of a primitive topology from which vertex attributes that are constant across the primitive are taken.
Quad Quad word (QQword)	QQ	A fundamental data type, QQ represents 32 bytes.
Quad Word (QWord)	QW	A fundamental data type, QW represents 8 bytes.
Rasterization	—	Conversion of an object represented by vertices into the set of pixels that make up the object.
Region-based addressing	—	Collective term for the register addressing modes available in the EU instruction set that permit discontiguous register data to be fetched and used as a single operand.
Render Cache	RC	Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test.
Render Target	RT	A destination surface in memory where render results are written.
Render Target Array Index	—	Selector of which of several render targets the current operation is targeting.
Root Thread	—	A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Sampler	—	Shared function that samples textures and reads data from buffers on behalf of EU programs.
Scratch Space	—	Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc.
Shader	—	A Gen4 program that is supplied by the application in an high level shader language, and translated to Gen4 instructions by the jitter.
Shared Function	SF	Function unit that is shared by EUs. EUs send messages to shared functions; they consume the data and may return a result. The Sampler, Data Port and Extended Math unit are all shared functions.

Term	Abbr.	Definition
Shared Function ID	SFID	Unique identifier used by kernels and shaders to target shared functions and to identify their returned messages.
Single Instruction Multiple Data	SIMD	The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at instruction level. It can also be used to describe the instructions in such architecture.
Source	—	Describes an input or read operand
Spawn	—	To initiate a thread for execution on an EU. Done by the thread spawner as well as most FF units in the 3D pipeline.
Sprite Point	—	Point object using full range texture coordinates. Points that are not sprite points use the texture coordinates of the point's center across the entire point object.
State Descriptor	—	Blocks in memory that describe the state associated with a particular FF, including its associated kernel pointer, kernel resource allowances, and a pointer to its surface state.
State Register	SR	The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP.
State Variable	SV	An individual state element that can be varied to change the way given primitives are rendered or media objects processed. On Gen4 state variables persist only in memory and are cached as needed by rendering/processing operations except for a small amount of non-pipelined state.
Stream Output	—	A term for writing the output of a FF unit directly to a memory buffer instead of, or in addition to, the output passing to the next FF unit in the pipeline. Currently only supported for the Geometry Shader (GS) FF unit.
Strips and Fans	SF	Fixed function unit whose main function is to decompose primitive topologies such as strips and fans into primitives or objects.
Sub-Register	—	Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, *r2*, is 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*.
Subsystem	—	The Gen4 name given to the resources shared by the FF units, including shared functions and EUs.
Surface	—	A rendering operand or destination, including textures, buffers, and render targets.
Surface State	—	State associated with a render surface including
Surface State Base Pointer	—	Base address used when referencing binding table and surface state data.

Term	Abbr.	Definition
Synchronized Root Thread	—	A root thread that is dispatched by TS upon a 'dispatch root thread' message.
System IP	SIP	There is one global System IP register for all the threads. From a thread's point of view, this is a virtual read-only register. Upon an exception, hardware performs some bookkeeping and then jumps to SIP.
System Routine	—	Sequence of Gen4 instructions that handles exceptions. SIP is programmed to point to this routine, and all threads encountering an exception will call it.
Thread	—	An instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction – a send instruction with EOT that signals the thread termination. Threads in GEN4 system may be independent from each other or communicate with each other through Message Gateway share function.
Thread Dispatcher	TD	Functional unit that arbitrates thread initiation requests from Fixed Functions units and instantiates the threads on EUs.
Thread Identifier	TID	The field within a thread state register (SR0) that identifies which thread slots on an EU a thread occupies. A thread can be uniquely identified by the EUID and TID.
Thread Payload	—	Prior to a thread starting execution, some amount of data will be pre-loaded in to the thread's GRF (starting at r0). This data is typically a combination of control information provided by the spawning entity (FF Unit) and data read from the URB.
Thread Spawner	TS	The second and the last fixed function stage of the media pipeline that initiates new threads on behalf of generic/media processing.
Topology	—	See Primitive Topology.
Unified Return Buffer	URB	The on-chip memory managed/shared by GEN4 Fixed Functions in order for a thread to return data that will be consumed either by a Fixed Function or other threads.
Unsigned Byte integer	UB	A numerical data type of 8 bits.
Unsigned Double Word integer	UD	A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.
Unsigned Word integer	UW	A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.
Unsynchronized Root Thread	—	A root thread that is automatically dispatched by TS.
URB Dereference	—	See URB Reference
URB Entry	UE	URB Entry: A logical entity stored in the URB (such as a vertex), referenced via a URB Handle.

Term	Abbr.	Definition
URB Entry Allocation Size	—	Number of URB entries allocated to a Fixed Function unit.
URB Fence	Fence	Virtual, movable boundaries between the URB regions owned by each FF unit.
URB Handle	—	A unique identifier for a URB entry that is passed down a pipeline.
URB Reference	—	For the most part, data is passed down the fixed function pipeline in an indirect fashion. The data is typically stored in the URB and accessed via a URB handle. When a pipeline stage passes the handle of a URB data entry to a downstream stage, it is said to make a URB reference. Note that there may be several references to the same URB data entry in the pipeline at any given time. When a downstream stage accesses the URB data entry via a URB handle, it is said to "dereference" the URB data entry. When there are no longer any references to a URB data entry within the pipeline, the URB storage can be reclaimed.
Variable Length Decode	VLD	The first stage of the video decoding pipe that consists mainly of bit-wide operations. GEN4 supports hardware VLD acceleration in the VFE fixed function stage.
Vertex Buffer	VB	Buffer in memory containing vertex attributes.
Vertex Cache	VC	Cache of Vertex URB Entry (VUE) handles tagged with vertex indices. See the VS chapter for details on this cache.
Vertex Fetcher	VF	The first FF unit in the 3D pipeline responsible for fetching vertex data from memory. Sometimes referred to as the Vertex Formatter.
Vertex Header	—	Vertex data required for every vertex appearing at the beginning of a Vertex URB Entry.
Vertex ID	—	Unique ID for each vertex that can optionally be included in vertex attribute data sent down the pipeline and used by kernel/shader threads.
Vertex Index	—	Offset (in vertex-sized units) of a given vertex in a vertex buffer. Available in the VF and VS units for debugging purposes. Not unique per vertex instance.
Vertex Sequence Number	—	Unique ID for each vertex sent down the south bus that may be used to identify vertices for debugging purposes.
Vertex Shader	VS	An API-supplied program that calculates vertex attributes. Also refers to the FF unit that dispatches threads to "shade" (calculate attributes for) vertices.
Vertex URB Entry	VUE	A URB entry that contains data for a specific vertex.
Vertical Stride	VertStride	The distance in element-sized units between 2 vertically-adjacent elements of a Gen4 region-based GRF access.

Term	Abbr.	Definition
Video Front End	VFE	The first fixed function in the GEN4 generic pipeline; performs fixed-function media operations.
Viewport	VP	Post-clipped geometry is mapped to a rectangular region of the bound rendertarget(s). This rectangular region is called a viewport. Typically, the viewport is set to the full extent of the rendertarget(s), but any subregion can be used as well.
Windower IZ	WIZ	Term for Windower/Masker that encapsulates its early ("intermediate") depth test function.
Windower/Masker	WM	Fixed function triangle/line rasterizer.
Word	W	A numerical data type of 16 bits, W represents a signed word integer.

§§

2 Subsystem Overview

2.1 Introduction

The Gen4 subsystem consists of an array of *execution units* (*EUs*, sometimes referred to as an arrray of *cores*) along with a set of *shared functions* outside the EUs that the EUs leverage for I/O and for complex computations. Programmers access the Gen4 Subsystem via the 3D or Media pipelines.

EUs are general-purpose programmable cores that support a rich instruction set that has been optimized to support various 3D API shader languages as well as media functions (primarily video) processing.

Shared functions are hardware units which serve to provide specialized supplemental functionality for the EUs. A shared function is implemented where the demand for a given specialized function is insufficient to justify the costs on a per-EU basis. Instead a single instantiation of that specialized function is implemented as a stand-alone entity outside the EUs and shared amongst the EUs.

Invocation of the shared functionality is performed via a communication mechanism call a "message". A message is a small, self-contained packet of information created by a kernel and directed to specific shared function. The message is defined by sequential series of MRF registers which hold message operands, a destination shared function ID, a function-specific encoding of the desired operation to be performed, and a destination GRF register to which any writeback response is to be directed. Messages are dispatched to the shared function under software control via the 'send' instruction. This instruction identifies the contents of the message and the GRF register location(s) to direct any response.

The message construction and delivery mechanisms are general in their definition and capable of supporting a wide variety of shared functions.

2.2 Subsystem Topology

The subsystem is organized as an array of EUs, and a set of functions that are shared among all of the EUs. (The EU array is further divided into rows with each row having its own first level instruction cache and Extended Math shared function, though this aspect of the implemented topology is not exposed to software). The Sampler, DataPort, URB and Message Gateway functions are shared among the entire array of EUs.

2.3 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

- debugging (see *Debugging*)
- 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. See the *Gen4 Core* volume for details such as EU registers and instruction set.

2.4 Thread Dispatching

When the 3D and Media pipelines send requests for thread initiation to the Subsystem, the thread Dispatcher receives the requests. The dispatcher performs such tasks as arbitrating between concurrent requests, assigning requested threads to hardware threads on EUs, allocating register space in each EU among multiple threads, and initializing a thread's registers with data from the fixed functions and from the URB. This operation is largely transparent to software.

To aid in debug, the thread dispatcher can be programmed by software to limit the number of EUs utilized from the maximum available in hardware down to as little as a single EU. It can also be programmed (independently from the number of EUs being utilized) to limit the number of threads that an EU will run concurrently down to as few as one. These features should not be required for normal use but will come in handy for debug. See the *Debugging* chapter for more information.

2.5 Shared Functions

In general, a shared function has the ability to receive messages at its input, perform some specialized amount of work for each, and if required, generate output back to the message's originating execution unit (Message Gateway may generate output to a target execution unit specified by the message).

To uniquely identify shared functions, each is assigned a unique 4-bit identifier code called its 'Function ID'. This ID is specified in the 'send' instruction's 32b <desc> field of each message. Gen4 Function ID assignments are listed in the *Graphics Processing Engine* chapter of this specification.

Each shared function may support one or more related operations within itself. For example an Extended Math shared function may support operations such as reciprocal, sine, cosine, and/or others. These are generically referred to as sub-functions. The communication method as to which sub-function is desired is typically contained in the 16b 'function-control' field of the 'send' instruction <desc> field. Alternatively, a function may choose to define sub-function encodings in-band within message payload, or in the case of a single function shared-function, the function code may be implied. The architecture, in no way interprets the sub-function code and the actual implementation choice is left to the function itself.

The Shared Function units included in the Subsystem are as follows (refer to the chapters devoted to each of these functions):

- Extended Math function
- Sampling Engine function
- DataPort function
- Message Gateway function
- Unified Return Buffer (URB)
- Thread Spawner (TS)
- Null function

The **Extended Math** function acts as an extension of the math functions already available inside the EUs. Certain functions such as inverse, square root, exponentiation, etc., require significant hardware resources to implement and are used infrequently enough that it is inefficient to implement them separately in each EU. The EUs therefore send the operands for these operations along with the operation to be performed to the Extended Math function which computes and returns the result to the requesting EU.

The **Sampling Engine** acts a (read-only) I/O port on behalf of the EUs, translating texture coordinates (and/or structure references) to memory addresses, reading texels and/or other data from memory, and in the case of texels, combining and filtering them according to programmed state. The resulting pixel and/or other data are then returned to the requesting EU.

The **Data Port** function acts as another I/O port on behalf of the EUs. It is both a read and a write port, and the only way for the Graphics Processing Engine to write results (e.g., images) back to memory. The Data Port contains the render and depth caches which receive the newly rendered pixels and write them out to memory when necessary. They also permit previously rendered objects to be read back efficiently by the Graphics Processing Engine in order to blend them with other rendered objects and test for visibility of newly rendered objects. Finally, the Data Port also provides read access constant buffers (arrays of constants in memory.)

The **Message Gateway** allows a thread to communicate (send a message to) another thread. A key is used to connect the sender and receiver threads, and a simple gateway protocol is used to send messages. This is primarily intended for media where a parent/child thread model is sometimes used and requires parent and child threads to synchronize and efficiently share information. It is not intended to be used by 3D graphics rendering threads.

The **Unified Return Buffer** (URB) is a single set of registers that EU threads use to return result data for future fixed functions and their threads to make use of. Individual entries in the buffer are "owned" by a given fixed function but a mechanism is provided where other fixed functions (those that follow) can read the data placed there by another fixed function. The buffer is considered a "Shared Function" since EUs need to be able to write result data to it using messages. In general, EU threads write their final results either to memory via the Data Port or to the URB for re-use by subsequent EU threads or certain 3D pipeline fixed-function units (CLIP, GS).

The **Thread Spawner** (TS) is a Shared Function that acts as a conduit for dispatching kernel-software-generated threads, one thread can request another thread to be dispatched by sending a request to the TS. TS is unique as it is also a Fixed Function in the media pipeline for dispatching threads originated from Video Front End fixed function.

The **Null** shared function is supported to allow the broadcast of certain information (e.g, End Of Thread) without invoking any other operation or response.

2.6 Messages

Communication between the EUs and the shared functions and between the fixed function pipelines (which are not considered part of the "Subsystem") and the EUs is accomplished via packets of information called *messages*. Message transmission is requested via the 'send' instruction. Refer to the 'send' instruction definition in the *ISA Reference* chapter for details.

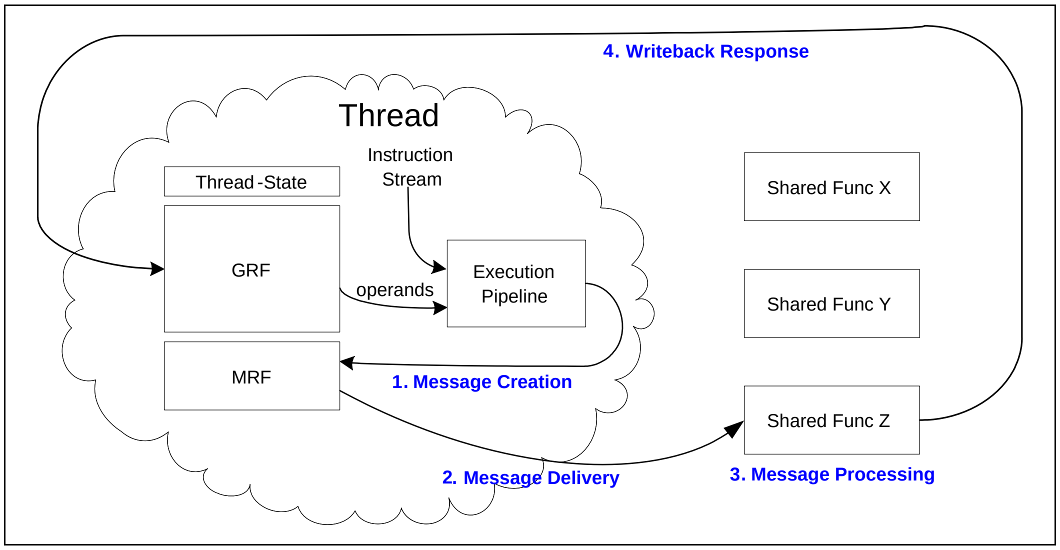
The information transmitted in a message falls into two categories:

- **Message Payload** data sourced from some number of registers (from 1 to 15 registers) in the Message Register File (MRF). The contents of the payload are dependent on the target function and specific function (*et al.*), and may contain a header portion and/or data portion.
- Associated ("sideband") information provided by:
  - **Message Descriptor** specified with the 'send' instruction. Included in the message descriptor is control and routing information such as the target function ID, message payload length, response length, etc.
  - Additional information provided by the 'send' instruction, e.g., the starting destination register number, the execution mask (EMASK), etc.
  - A small subset of Thread State, such as the Thread ID, EUID, etc.

The software view of messages is shown in Figure 2-1. There are four basic phases to a message's lifetime as illustrated below:

1.Creation	The thread assembles the message payload into the Message Register File (MRF). This is done by a series of one or more instruction which specify a MRF register as the destination.
2.Delivery	The thread issues the message for delivery via the 'send' instruction. The 'send' instruction specifies the MRF register which is the first of a sequential register series which makes the data payload, the length of the message payload within the MRF, the destination shared function ID (SFID), and where in the GRF any response is to be directed. The messaging subsystem will enqueue the message for delivery and eventually route the message to the specified shared function.
3.Processing	The shared function receives the message and services it accordingly, as defined by the shared function definition.
4.Writeback	If called for, the shared function delivers an integral number of registers of data to the thread's GRF in response to the message.

Figure 2-1. Data Flow Associated With Messages



2.6.1 Message Register File (MRF)

Each thread has a dedicated MRF which is logically identical to the GRF: 256 bits wide per register, with word-wide addressability. There are 16 MRF registers, referred to as "m0".."m15". From a software perspective, the MRF is write-only and thus may only be used as a destination specifier. Limited register-region specifications are allowed so long as the region is contained within a single MRF register.

Each register of the MRF has an associated in-flight status, indicating the contents of the register is needed as part of a pending message, but has yet to be transmitted by the hardware. This bit is set at the time the message is enqueued for delivery via the 'send' instruction. Should a subsequent write to an in-flight register be attempted, the execution unit will temporarily suspend the thread's execution until the register's in-flight status is cleared (i.e., the message has been transmitted).

Register m0 is reserved for System Routine (exception handling and debug) purposes, thus normal threads should construct their messages in m1..m15. The thread is free to start a message payload at any MRF register location, even to the point of having multiple messages under construction at the same time in non-overlapping spaces in the MRF. Further multiple messages over non-overlapping MRF space can be enqueued awaiting transmission at the same time. Regardless of actual hardware implementation, the thread should not assume that MRF addresses above m15 wrap to legal MRF registers.

2.6.2 Send Instruction

Messages are sent programmatically by the thread through the 'send' instruction. This instruction enqueues a message for delivery and marks as in-flight all MRF registers used for the message payload. It also allows for an optional implied move of one GRF register to a MRF register prior to the message being issued. This implied move allows for a higher message performance, eliminating the explicit 'mov' that would normally be required to move R0 to the lead MRF register of the message (as required by many message definitions).

A typical 'send' instruction is exemplified here (please see the ISA for a full instruction description). This example performs an implicit move from r0 to m3, then issues a message to the Extended Math unit, with a payload of 1 register starting at m3, and expecting 1 register in reply to be placed in r5.

```
send (16) r5 m3 r0 0x01110001
```

The execution unit guarantees that any prior instruction which wrote to a MRF register is guaranteed to have retired, and its result written to the destination MRF register in time for message transmission.

2.6.3 Creating and Sending a Message

A code snippet is listed below, showing a 4-register message (m3 to m6) whose response is directed to r30. Note that message construction does not have to occur in MRF register order.

```
...
mul  (8)   m4     r20      r19
mov  (8)   m6     r21
add  (8)   m5     r29      r28
send  (8)  r30    m3       r0   <desc>
...
```

Once a 'send' instruction is issued, the MRF registers used for its payload are marked as 'in-flight'. These registers remain in this state until the message is actually transmitted to the shared function and the register contents are no longer need. Any subsequent write to a MRF register which is in-flight results in a dependency and a thread switch until such time that the in-flight condition is cleared. An example is shown below in which the attempt to re-use m6 may result in a thread switch until message 1 is transmitted.

```
...
// --- message 1 ---
mul  (8)   m4     r20      r19
mov  (8)   m6     r21
add  (8)   m5     r29      r28
send  (8)  r30    m3       r0   <desc>
...


// --- message 2 ---
mov  (8)   m6     r15      // thread switch until the
                           // previous msg is sent and
                           // m6 in-flight is cleared.
...
```

MRF registers of one message may be reused for a subsequent message without restriction. The in-flight check mechanism prevents a MRF register staged as part of a pending message from being altered while awaiting transmission. Further, a thread may rely on the contents of a MRF register being unaltered after message transmission. This allows the thread to quickly issue an identical or slightly altered message using the same MRF register set without having to re-construct the entire payload.

Although more than one message may be enqueued at any point in time, care must be taken by the programmer to ensure that each message's destination GRF register region, if any, does no over lap with that of another enqueued message. This condition is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles in the current implementation, the outcome in the GRF is indeterminate; It may be the result from the first message, or the result from the second message, or a mixture of data from both.

2.6.4 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*). It contains the debug fields (fixed at DW6 and DW7) and state fields (such as binding table pointer, sampler state pointer, etc.) following a consistent format structure. 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.

2.6.5 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).

2.6.6 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).

2.6.7 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.

2.6.8 End-Of-Thread (EOT) Message

The final instruction of all threads must be a 'send' instruction which signals 'End-Of-Thread' (EOT). An EOT message is one in which the EOT bit is set in the 'send' instruction's 32b <desc> field. When issuing instructions, the EU looks for an EOT message, and when issued, shuts down the thread from further execution and considers the thread completed.

Only a subset of the shared functions can be specified as the target function of an EOT message, as shown in the table below.

Target Shared Functions supporting EOT messages	**Target Shared Functions not supporting EOT messages**
Null, DataPortWrite, URB, MessageGateway, ThreadSpawner	DataPortRead, Sampler

Both the fixed-functions and the thread dispatcher require EOT notification at the completion of each thread. The thread dispatcher and fixed functions in the 3D pipeline obtain EOT notification by snooping all message transmissions, regardless of the explicit destination, looking for messages which signal end-of-thread. The Thread Spawner in the media pipeline does not snoop for EOT. As it is also a shared function, all threads generated by Thread Spawner must send a message to Thread Spawner to explicity signal end-of-thread.

The thread dispatcher, upon detecting an end-of-thread message, updates its accounting of resource usage by that thread, and is free to issue a new thread to take the place of the ended thread. Fixed functions require end-of-thread notification to maintain accounting as to which threads it issued have completed and which remain outstanding, and their associated resources such as URB handles.

Unlike the thread dispatcher, fixed-functions discriminate end-of-thread messages, only acting upon those from threads which they originated, as indicated by the 4b fixed-function ID present in R0 of end-of-thread message payload. This 4b field is attached to the thread at new-thread dispatch time and is placed in its designated field in the R0 contents delivered to the GRF. Thus to satisfy the inclusion of the fixed-function ID, the typical end-of-thread message generally supplies R0 from the GRF as the first register of an end-of-thread message.

As an optimization, an end-of-thread message may be overload upon another “productive” message, saving the cost in execution and bandwidth of a dedicated end-of-thread message. Outside of the end-of-thread message, most threads issue a message just prior to their termination (for instance, a 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.

2.6.9 Performance

The Gen4 Architecture imposes no requirement as to a shared function’s latency or throughput. Due to this as well as factors such as message queuing, shared bus arbitration, implementation choices in bus bandwidth, and instantaneous demand for that function, the latency in delivering and obtaining a response to a message is non-deterministic. It is expected that a Gen4 implementation has some notion of fairness in transmission and servicing of messages so as to keep latency outliers to a minimum.

Other factors to consider with regard to performance:

- A thread may choose to have multiple messages under construction in non-overlapping registers the MRF at the same time.
- Multiple messages are allowed to be enqueued for transmission at the same time, so long as their MRF payload registers do not overlap.
- Messages may rely on the MRF registers being maintained across a send message, thus constructing subsequent messages overlaid on portions of a previous message,
- Software prefetching techniques may be beneficial for long latency data fetches (i.e., issue a load early in the thread for data that is required late in the thread).

2.6.10 Message Description Syntax

All message formats are defined in terms of DWords (32 bits). The message registers in all cases are 256 bits wide, or 8 DWords. The registers and DWords within the registers are named as follows, where n is the register number, and d is the DWord number from 0 to 7, from the least significant DWord at bits [31:0] within the 256-bit register to the most significant DWord at bits [255:224], respectively. For writeback messages, the register number indicates the offset from the specified starting destination register.

Dispatch Messages: **R**n.d

Dispatch messages are sent by the fixed functions to dispatch threads. See the fixed function chapters in the *3D and Media* volume.

SEND Instruction Messages: **M**n.d

These are the messages initiated by the thread via the SEND instruction to access shared functions. See the chapters on the shared functions later in this volume.

Writeback Messages: **W**n.d

These messages return data from the shared function to the GRF where it can be accessed by thread that initiated the message.

The bits within each DWord are given in the second column in each table.

2.6.11 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 (see the *Debugging* chapter for details). The set of possible errors is listed in Table 2-1 with the associated outcome. Please see the chapters on debug and error handling for detailed information.

Table 2-1. Error Cases

Error	Outcome
Bad Shared Function ID	The message is discarded before reaching any shared function. If the message specified a destination, those registers will be marked as in-flight, and any future usage by the thread of those registers will cause a dependency which will never clear, resulting in a hung thread and eventual time-out.
Unknown opcode Incorrect message length	The destination shared function detects unknown opcodes (as specified in the 'send' instructions <desc> field), and known opcodes where the message payload is either too long or too short, and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through and interrupt mechanism (details tbd), then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.
Incorrect response length	Case: too few registers specified – the thread may proceed with execution prior to all the data returning from the shared function, resulting in the thread operating on bad data in the GRF. Case: too many registers specified – the message response does not clear all the registers of the destination. In this case, if the thread references any of the residual registers, it may hand and result in an eventual time-out.
Improper use of End-Of-Thread (EOT)	Any 'send' instruction which specifies EOT must have a 'null' destination register. The EU enforces this and, if detected, will not issue the 'send' instruction, resulting in a hung thread and an eventual time-out. The 'send' instruction specifies that EOT is only recognized if the <desc> field of the instruction is an immediate. Should a thread attempt to end a thread using a <desc> sourced from a register, the EOT bit will not be recognized. In this case, the thread will continue to execute beyond the intended end of thread, resulting in a wide range of error conditions.
Two outstanding messages using overlapping GRF destinations ranges	This is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles, the outcome in the GRF is indeterminate; may be the result from the first message, or the result from the second message, or a combination of both.

3 Debugging

3.1 Introduction

The Gen4 Architecture includes dedicated logic to facilitate debug of the system. Each fixed function unit contains logic to allow trapping data associated with a specific element (vertex, polygon, pixel, etc.). This logic will be enabled and controlled via debug registers in MMIO space.

Most units output a debug tag along with the vertices they output. These tags contain the FFID of the unit, a thread ID that increments with each thread dispatched by the unit (if the unit dispatches threads), and a vertex sequence number that uniquely identifies each output. These sequence numbers are thread-relative for units whose outputs are generated by threads. These debug tags are available to the next enabled unit as optional trap data.

In addition, a mechanism is provided to allow setting breakpoints in selected threads as they are executing in the core. The breakpoints can be set at an instruction level or via an MMIO register using a physical thread address or on a certain opcode. When a breakpoint is encountered, the execution unit will switch to a system routine that can be used to write out internal data (GRF, MRF, thread state).

The trapping of data associated with a specific element can be used to enable the breakpoint mechanism such that only the thread associated with the trapped element will switch to the system routine at the breakpoint; other threads executing the same instructions on different elements will not have the breakpoint enabled. Alternatively, breakpoints can be enabled for all threads initiated by a given FF unit. A global flag is also available that will cause all threads (from all FF units) to have breakpoints enabled.

The Shared Functions will also have some debug controls, such as running in a non-pipelined mode of operation.

These debug features are described in more detail below.

3.2 The Snapshot Mechanism

All of the fixed function units (and some of the shared function units) have "snapshot" debugging capability. Each fixed function has a debug control register with (nominally) 5 control bits that are associated with the function, plus a trigger value register, and a debug data register where the captured "snapshot" can be read.

Size in Bits	Name	Description
1	Snapshot Enable	Enables the FF unit to capture trap data based on a match with the Debug Snapshot Trigger Value.
8	Snapshot Output Mux Select	Controls which DW of trap data is visible. The number of valid values for this field varies by FF unit and is generally much less than the 256 maximum allowed by the field. Note that all data defined in this field is latched – the value in this field only determines which DW is currently visible in
1	Snapshot Complete	This bit is set to 1 by hardware when a snapshot compare succeeds. This bit should be set to 0 via MMIO write after reading the desired snapshot return values in order to enable another snapshot to occur.
1	Thread Snapshot Enable	When set, debug will be enabled on the thread that is associated with the trapped element.
1	Snapshot All Threads	When set, debug will be enabled for all threads generated by this Fixed Function Unit. Setting this bit means **Thread Snapshot Enable** is effectively ignored.
N	Debug Snapshot Trigger Value	The value in this field is compared with a corresponding value associated with each "element" processed by the FF unit. In most cases the value associated with each element is an incrementing "thread ID" but in some cases it is some other value associated with an element processed by the FF unit.
32	Debug Data	Once a snapshot has been triggered by a successful compare, the DW of data indicated by **Snapshot Output Mux Select** can be read here.

Some of the fixed/shared functions have additional fields that can be used to further qualify the snapshot comparison. See the following sections on each FF unit for unit-specific control capabilities.

3.2.1 Debug Trigger Counters

Fixed function units contain various counters whose values can be compared against the Debug Snapshot Trigger Value of the unit, captured by the next enabled unit in the pipeline, or used within the unit to eliminate some elements from processing in order to minimize test cases.

Fixed Function Unit	Available Counters
VF	Vertex Sequence Number Primitive Topology Number (for test minimization only)
VS	Thread ID
GS	Thread ID
Clipper	Thread ID
SF	Thread ID Primitive Sequence Number
WM	Thread ID Primitive Sequence Number (same as the one in SF)

These counters are reset to 0 by a issuing an MI_FLUSH with the **Global Snapshot Counter Reset** bit set. They count up continuously until they are reset again (or until they roll over.) It is recommended that these counters be reset no less often than once per frame to avoid rollover.

The counters are controlled by a global debug enable which is set from a Command Stream MMIO register (see Memory Interface Registers chapter). The counters will only increment if the global debug enable is set.

The SVG unit will qualify the debug enable signals to all fixed function and shared funciton units with the global debug enable (as described in each of the following sections).

3.3 Fixed Function Debug Process

3.3.1 Overview

Each fixed function unit has a unique (4-bit) code as a unit identifier defined. Each unit will generate an incrementing output identifier for each output it generates. For each thread dispatched by a fixed function unit, an incrementing thread ID will be generated and passed along to the execution units.

In order to make effective use of the debug logic, a failing test case should be repeatable. Once a repeatable failure is created, the debug mechanisms described in this chapter can be used to facilitate debug.

The first step of debug is to identify the screen coordinates of a failing pixel (x,y location). This pixel location is placed in WIZ – Debug Snapshot Trigger Value. The Windower snapshot is then enabled using WIZ – Debug Control.

The driver then needs to monitor WIZ – Debug Control bit 31 to determine when the snapshot operation has been completed. When this bit becomes set by the hardware, the driver can then read back data that was captured by the snapshot operation. This data is read from WIZ – Debug Return Data. The selection of which data to read is made using the snapshot select field of WIZ – Debug Control.

If bit 1 of WIZ – Debug Control is set, the snapshot flag will be passed to the Thread Dispatcher. If breakpoints are enabled (see Thread Dispatcher), a breakpoint will be enabled for the Pixel Shader thread that generated the pixel of interest. Various types of breakpoints can be programmed (see Attention Signaling from EU to Host).

If the pixel computations seem correct, or the corruption covers an entire polygon, the Primitive Sequence Number read back from the Windower snapshot can be used to snapshot computations in the SF unit.

By enabling the SF snapshot (SF – Debug Control), and setting the snapshot ID (SF – Debug Snapshot Trigger Value) to the count of the primitive of interest, a snapshot will occur in the SF unit when that primitive is processed.

The mechanism for determining that the snapshot is complete is the same as described for the Windower above. Various data can be read using the snapshot select for SF. Enabling the thread snapshot will allow breakpoints to be used to debug the Setup kernel that generated the coefficient data for the selected primitive.

If the corruption seems to be related to one of the input vertices to SF, the snapshot select can be used to read back the debug ID of the input vertices that create the primitive. This debug data can then be used in the unit that sourced the vertex to the SF (identified by the FF ID of the vertex debug field).

If the source of the vertex data is the Clipper, the thread ID field of the debug ID can be used to program a Clipper snapshot (see Clipper). The Clipper Thread can be debugged by enabling the thread snapshot. The input vertices to the clipper can be examined and used to trace the input further upstream to the Geometry Shader or Vertex Shader.

Similar debug controls are used to snapshot Geometry Shader (see Geometry Shader) and Vertex Shader (see Vertex Shader) outputs.

3.3.2 Vertex Shader Debug

Vertex Shader debug is somewhat unique. Some additional steps are required to identify the Vertex Shader thread for a specific vertex output. Each output vertex has a unique sequence number, however multiple outputs can map to the same vertex shader thread. This is due to the fact that this unit has a vertex cache. A vertex can be shaded once then used multiple times.

Given a reproducible test case, one can identify the thread that shaded a given vertex. Assuming an errant vertex has been discovered in GS, the vertex sequence number from the previous enabled FF can be determined from the GS snapshot. The associated FFID captured in GS will indicate that the vertex came either from the VS unit or (when VS is in bypass mode) directly from the VF unit. If the vertex came from the VF unit then the VS is not involved.

If VS is the identified unit, the test must now be rerun with the *VF* snapshot trigger value programmed to that vertex sequence number. The snapshot obtained from that test run yields the thread ID of the thread that shaded that vertex, regardless of whether the vertex cache is enabled.

The test must now be rerun once more with the VS snapshot trigger value set to the thread ID of the offending thread, which can now be debugged.

3.3.3 SVG Debug

The State Variable unit contains base addresses which nearly all 3D/Media accesses are offset from. These registers allow the base addresses to be captured for inspection. The base addresses will be latched if they change while **Global Debug Enable** in the INSTPM register is set. This way the base addresses for the context being debugged can be read here anytime, even if another context is currently executing.

3.3.3.1 SVG_CTL —Debug Control

Address Offset:	07400h–7403h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:8	**Debug Output Mux Select.** Controls which 32 bits of the debug data are returned 0 = return General State Base Address for the context which has debug enabled 1 = return Surface State Base Address for the context which has debug enabled 2 = return Indirect Object Base Address for the context which has debug enabled 3 = return General State Access Upper Bound for the context which has debug enabled 4 = return Indirect Indirect Object Access Upper Bound for the context which has debug enabled 5 = return System Instruction Pointer for the context which has debug enabled 6 - 255 = Documented in an SVG specific document outside of the PRM.
7:0	Reserved : MBZ

3.3.3.2 SVG_RDATA—Debug Return Data

Address Offset:	07404h–7407h
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**SVG Debug Data.** Returns data based on debug output mux select

3.3.3.3 SVG_WORK_CTL—Debug Workaround Control

Address Offset:	07408h–740Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:5	Reserved : MBZ
4	Reserved. **[DevBW-A,B]** DAP Stateless Access ECO: “0” –Indicates Stateless Accesses disallowed “1” – Inicates Stateless accesses allowed
3:2	Reserved : MBZ
1	**Disable WIZ Panic Dispatch** **If set, disables the B0 BW modification which forces the WIZ to dispatch partial payloads for certain non-promoted cases** 0 = normal operation (force partial dispatch for potential depth cache deadlock cases) 1 = don’t perform deadlock check for non-promotable cases
0	**YUV 4:2:2 Chrominace Mode.** This field controls whether the chrominance for odd pixels is computed by an interpolation between the adjacent even pixels, or a replication from the pixel to the left. **Programming Notes:** • the texture caches must be invalidated after switching the state of this bit 0 = Replication 1: Interpolation

3.3.4 Vertex Fetch

The Vertex Fetch unit will output the VF FFID and a vertex index with each vertex it outputs. The VF will have the ability to snapshot any given output vertex index or a vertex sequence number. When a snapshot compare succeeds, the VF will latch a number of internal signals which can be examined by setting the VF **Snapshot Output Mux Select** appropriately and examining the **VF Debug Return Data** register. The thread ID of the VS thread that shades the vertex can also be captured here; VS will be effectively bypassed for a given vertex if its shaded results are already available in the vertex cache.

The VF will also have the ability to process only a selected range of primitive topologies (3DPRIM commands) and a selected range of vertices for those primitives. This will allow a test to be minimized for debug purposes.

3.3.4.1 VF_CTL—Debug Control

Address Offset:	7500h–7503h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned 0 = return thread ID of vertex shader dispatch (**Snapshot Type** should be set to 0) 1 = Internal VF debug data (**Snapshot Type** should be set to 1) Defined in a VF-specific document outside the PRM
7:5	Reserved : MBZ
4	**Snapshot Type.** Controls whether the **Debug Snapshot Trigger Value** is compared against a 24-bit vertex sequence number or a vertex index (up to 32 bits). 0 = 24-bit Trigger Value is compared against the vertex sequence number (used to obtain thread ID) 1 = 32-bit Trigger Value is compared against the vertex index (used for all other VF debug snapshots)
3	**Skip Initial Primitives.** If set, the number of primitives (3DPRIM commands) programmed in the **Debug Starting Primitives Skipped** register will be parsed by the command streamer and then immediately discarded by the Vertex Fetch unit before any processing is done (qualified in SVG with global debug enable)
2	**Max Primitives Limit Enable.** If set, primitives (3DPRIM commands) beyond the number programmed in the **Debug Max Primitives** register will be parsed by the command streamer and then immediately discarded by the Vertex Fetch unit before any processing is done (qualified in SVG with global debug enable).
1	**Vertex Range Limit Enable.** If set, parameters in 3DPRIM commands will be overridden such that only vertices within a range specified by **Start Vertex Location Override** and **Vertex Count Per Instance Override** will be fetched and processed (qualified in SVG with global debug enable).
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the VF (qualified in SVG with global debug enable).

3.3.4.2 VF_STRG_VAL—Debug Snapshot Trigger Value

Address Offset:	7504h–7507h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:0	**Vertex Sequence Number or Vertex Index.** When Snapshot Type is 0, bits 23:0 of this field will be compared to the sequence number of the current vertex being processed; bits 31:24 MBZ. When Snapshot Type is 1, this field will be compared to the index of the current vertex being processed. Bits 31:16 and bits 31:8 MBZ for Word and Byte indices, respectively. When a match occurs, signals within the FF unit will be latched for read back using the **VF Debug Return Data** register and the Snapshot Mux Select field of **VF Debug Control**.

3.3.4.3 VF_STR_VL_OVR —Debug Start Vertex Location Override

Address Offset:	7508h–750Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:0	**Start Vertex Location Override.** Overrides the **Start Vertex Location** of all primitives when **Vertex Range Limit** is enabled.

3.3.4.4 VF_VC_OVR —Debug Vertex Count Override

Address Offset:	750Ch–750Fh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:0	**Vertex Count Per Instance Override.** Overrides the **Vertex Count Per Instance** of all primitives when **Vertex Range Limit** is enabled.

3.3.4.5 VF_STR_PSKIP —Debug Starting Primitives Skipped

Address Offset:	7510h–7513h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
23:0	**Starting Primitives Skipped.** If **Skip Initial Primitives** is enabled, this field specifies the number of primitives (3DPRIM commands) that should be skipped prior to beginning normal primitive processing.

3.3.4.6 VF_MAX_PRIM —Debug Max Primitives

Address Offset:	7514h–7517h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved. MBZ
23:0	**Max Primitives.** If **Max Primitive Limit Enable** is set, this field specifies the maximum number of primitives (3DPRIM commands) that will be processed normally after which all succeeding primitives will be skipped. Note that primitives skipped due to enabling **Skip Initial Primitives** <u>are still counted</u> toward this limit.

3.3.4.7 VF_RDATA —Debug Return Data

Address Offset:	7518h–751Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Vertex Fetch Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.3.5 Vertex Shader

The Vertex Shader (VS) will output a unique identifier with each vertex it outputs. The four MSBs of this field will be the VS FFID code and the next 24 bits will be the unique vertex sequence ID.

VS will assign a unique ID to each thread dispatched. The MSB of R0.7 holds the snapshot flag. Following the snapshot flag will be 31 reserved bits. The thread ID will be passed to the thread dispatcher as part of the r0.6 header debug data.

R0.7	
SS Flag	Reserved
1 bit	31 bits

R0.6	
Reserved for SW Debug	**Thread ID**
8 bits	24 bits

VS will have snapshot logic which will generate a compare flag based on a match to a specific thread ID. As VS issues that thread to be dispatched, it will pass along the snapshot flag to the Thread Dispatcher (TD).

3.3.5.1 VS_CTL —Debug Control

Address Offset:	7600h–7603h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned. Vertex 0 and vertex 1 refer to the 2 vertices dispatched to a VS thread. No FFID is included in VS snapshot data since the only possible sourcing unit is VF (which cannot be bypassed.) 0 = Vertex 0 Index 1 = Vertex 1 Index 2 = Valid vertex count (Range 1-2) 3 = VS Kernel Pointer 4 - 255 = = Defined in a Vertex Shader specific document outside the PRM
7:3	Reserved : MBZ
2	**Snapshot All Threads.** If set, the snapshot flag will be set for all threads generated by this Fixed Function Unit (Overrides Snapshot Enable bit). If set, the data for the last thread executed will be captured in the FF unit. (qualified in SVG with global debug enable).
1	**Thread Snapshot Enable.** This bit is set to enable passing the snapshot flag into the geometry shader thread dispatch. If Snapshot All Threads is also set, the snapshot flag will be passed into the thread dispatch for every thread. (qualified in SVG with global debug enable).
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the VS (qualified in SVG with global debug enable).

3.3.5.2 VS_STRG_VAL—Debug Snapshot Trigger Value

Address Offset:	7604h–7607h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved : MBZ
23:0	**Thread ID Compare Value.** This field will be used by the FF logic to compare to the current thread being dispatched. When a match occurs, a snapshot flag will be generated and optionally passed into the thread dispatch. Various signals within the FF unit will also be latched for read back via MMIO registers.

3.3.5.3 VS_RDATA —Debug Return Data

Address Offset:	7608h–760Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**VS Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

Each output vertex will be tagged with a unique identifier. This identifier will consist of a 4-bit Fixed Function Unit ID, a 24-bit thread ID and an 8-bit relative vertex ID (VS only needs a 1-bit relative vertex ID, but this is a generic identifier field). The 1-bit thread relative ID should be generated by the VS fixed function logic in the output path.

3.3.6 Geometry Shader

The Geometry Shader (GS) will output a unique identifier with each vertex it outputs. The four MSBs of this field will be the GS FFID code, the next 24 bits will be the unique thread ID, and the 10 LSBs will be the thread relative vertex output (created at the output of the GS). If the GS is disabled, it will output zero for the thread relative vertex.

The thread ID will also be passed to the thread dispatcher in the r0.6 header. .

R0.7	
SS Flag	Reserved
1 bit	31 bits

R0.6	
Reserved for SW Debug	**Thread ID**
8 bits	24 bits

The GS will have snapshot logic which will generate a compare flag based on a match to a specific thread ID. As the GS issues that thread to be dispatched, it will pass along the snapshot flag to the Thread Dispatcher (TD).

3.3.6.1 GS_CTL —Debug Control

Address Offset:	7900h–7903h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Selects what data will be accessed in **GS Debug Data**: 0 = input vertex 0 FF ID in bits 31:28, Vertex 0 Sequence Number in bits 23:0 1 = reserved 2 = input vertex 1 FF ID in bits 31:28, Vertex 1 Sequence Number in bits 23:0 3 = reserved 4 = input vertex 2 FF ID in bits 31:28, Vertex 2 Sequence Number in bits 23:0 5 = reserved 6 = input vertex 3 FF ID in bits 31:28, Vertex 3 Sequence Number in bits 23:0 7 = reserved 8 = input vertex 4 FF ID in bits 31:28, Vertex 4 Sequence Number in bits 23:0 9 = reserved 10 = input vertex 5 FF ID in bits 31:28, Vertex 5 Sequence Number in bits 23:0 11 = reserved 12 = valid input vertex count (range 1-6) 13 = GS Kernel Pointer 14 – 255 = = Defined in a Geometry Shader-specific document outside the PRM
7:3	Reserved : MBZ
2	**Snapshot All Threads.** If set, the snapshot flag will be set for all threads generated by this Fixed Function Unit (Overrides Snapshot Enable bit). If set, the data for the last thread executed will be captured in the FF unit (qualified in SVG with global debug enable).
1	**Thread Snapshot Enable.** This bit is set to enable passing the snapshot flag into the geometry shader thread dispatch. If Snapshot All Threads is also set, the snapshot flag will be passed into the thread dispatch for every thread (qualified in SVG with global debug enable).
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the GS (qualified in SVG with global debug enable).

3.3.6.2 GS_STRG_VAL —Debug Snapshot Trigger Value

Address Offset:	7904h–7907h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Snapshot ID.** This field will be used by the FF logic to compare to the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. Various signals within the FF unit will also be latched for read back via MMIO registers.

3.3.6.3 GS_RDATA —Debug Return Data

Address Offset:	7908h–790Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**GS Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.3.7 Clipper

The Clipper will output a unique identifier with each vertex it outputs. The four MSBs of this field will be the Clipper FFID, the next 24 bits will be the thread ID, and the LSBs will be a sequential count of output vertices (incremented by the clipper fixed function output logic).

The thread ID will also be passed through to the Thread Dispatcher as part of the r0 header.

R0.7	
SS Flag	Reserved
1 bit	31 bits

R0.6	
Reserved for SW Debug	**Thread ID**
8 bits	24 bits

The Clipper will have snapshot logic which will generate a compare flag based on a match to a specific thread ID. As the Clipper issues that thread to be dispatched, it will pass along the snapshot flag to the Thread Dispatcher (TD).

The Clipper will also have the capability to snapshot vertices that are not clipped, but passed through from the up stream units, so that fixed function clip logic can be debugged (like clip test results).

3.3.7.1 CL_CTL—Debug Control

Address Offset:	7A00h–7A03h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30	Reserved. MBZ
29:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Selects what data will be accessed in **Clipper Debug Data**: 0 = Input Vertex 0 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 (Sequence Number for VF/VS) 1 = Input Vertex 0 Relative Vertex Count (0 if GS is disabled) 2 = Input Vertex 1 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 3 = Input Vertex 1 Relative Vertex Count (0 if GS is disabled) 4 = Input Vertex 2 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 5 = Input Vertex 2 Relative Vertex Count (0 if GS is disabled) 6 = Valid Vertex Count (Range 1-3) 7 = Clipper Kernel Pointer 8 – 255 = = Defined in a Clipper specific document outside the PRM
7:3	Reserved : MBZ
2	**Snapshot All Threads.** If set, the snapshot flag will be set for all threads generated by this Fixed Function Unit (Overrides Snapshot Enable bit). If set, the data for the last thread executed will be captured in the FF unit (qualified in SVG with global debug enable).
1	**Thread Snapshot Enable.** This bit is set to enable passing the snapshot flag into the geometry shader thread dispatch. If Snapshot All Threads is also set, the snapshot flag will be passed into the thread dispatch for every thread (qualified in SVG with global debug enable).
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the Clipper (qualified in SVG with global debug enable).

3.3.7.2 CL_STRG_VAL—Debug Snapshot Trigger Value

Address Offset:	7A04h–7A07h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Snapshot ID.** This field will be used by the FF logic to compare to the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. Various signals within the FF unit will also be latched for read back via MMIO registers.

3.3.7.3 CL_RDATA—Debug Return Data

Address Offset:	7A08h–7A0Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Clipper Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.3.8 Strips Fans

SF will generate a unique primitive sequence number for each incoming primitive. This ID will be passed along to the setup kernel in the thread ID field, and also passed along to the Windower. This same primitive sequence number will be used as a thread ID for each setup thread that is dispatched.

R0.7	
SS Flag	Reserved
1 bit	31 bits

R0.6	
Reserved for SW Debug	**Thread ID**
8 bits	24 bits

SF can cull some polygons (backface, degenerate) so not every input will generate an output to the Windower. The primitive count will be incremented for all primitives, including the ones that are culled.

The SF will have the ability to snapshot any given primitive number. Based on the snapshot compare the SF will output to an MMIO register the vertex ids of the incoming vertices that created the primitive.

As the SF issues the thread for that primitive to be dispatched, it will pass along the snapshot flag to the Thread Dispatcher (TD).

To facilitate debug, the SF will contain logic to only process primitives within a certain range (specified through MMIO).

To support test minimization, a user clipping rectangle can be specified via MMIO registers. When enabled, the SF will only process polygons which fall inside the specified clip region.

The snapshot flag will be passed along to the Windower.

3.3.8.1 SF_CTL —Debug Control

Address Offset:	7B00h–7B03h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30	**Cull All.** When set to 1 SF drops all incoming primitives
29:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select** Controls which 32 bits of the trap data are returned 0 = Input Vertex 0 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 1 = Input Vertex 0 Relative Vertex Count (0 if GS is disabled) 2 = Input Vertex 1 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 3 = Input Vertex 0 Relative Vertex Count (0 if GS is disabled) 4 = Input Vertex 2 FF ID in bits 31:28, Thread ID (Vertex Sequence Number for VF/VS) in bits 23:0 5 = Input Vertex 0 Relative Vertex Count (0 if GS is disabled) 6 = Vertex count (range 1-3) 7 = SF Kernel Pointer 8 – 255 = = Defined in an SF-specific document outside the PRM
7:5	Reserved : MBZ
4	**Min / Max Primitive Range Enable** If set, primitives outside the specified min / max range are culled (qualified in SVG with global debug enable)
3	**Debug Clip Rectangle Enable** If set, the drawing rectangle is overloaded with the specified debug clip rectangle (qualified in SVG with global debug enable)
2	**Snapshot All Threads.** If set, the snapshot flag will be set for all threads generated by this Fixed Function Unit (Overrides Snapshot Enable bit). If set, the data for the last thread executed will be captured in the FF unit (qualified in SVG with global debug enable).
1	**Thread Snapshot Enable.** This bit is set to enable passing the snapshot flag into the geometry shader thread dispatch. If Snapshot All Threads is also set, the snapshot flag will be passed into the thread dispatch for every thread (qualified in SVG with global debug enable).
0	**Snapshot Enable** This bit is set to enable the fixed function snapshot logic in the SF (qualified in SVG with global debug enable).

3.3.8.2 SF_STRG_VAL—Debug Snapshot Trigger Value

Address Offset:	7B04h–7B07h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Snapshot ID.** This field will be used by the FF logic to compare to the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. Various signals within the FF unit will also be latched for read back via MMIO registers.

3.3.8.3 SF_MIN_PR_IND—Debug Minimum Primitive Index

Address Offset:	7B08h–7B0Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Minimum Primitive Index.** Sets the lower bound on primitives to be processed (if Min / Max primitive range is enabled).

3.3.8.4 SF_MAX_PR_IND—Debug Maximum Primitive Index

Address Offset:	7B0Ch–7B0Fh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Maximum Primitive Index.** Sets the upper bound on primitives to be processed (if Min / Max primitive range is enabled).

3.3.8.5 SF_CLIP_RMIN– Debug Clip Rectangle Minimum Coordinates

Address Offset:	7B10h–7B13h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	**Debug Clip Rectangle Minimum Y.** Sets the lower Y bound on pixels to be processed (if debug clip rectangle is enabled).
15:0	**Debug Clip Rectangle Minimum X.** Sets the lower X bound on pixels to be processed (if debug clip rectangle is enabled).

3.3.8.6 SF_CLIP_RMAX—Debug Clip Rectangle Maximum Coordinates

Address Offset:	7B14h–7B17h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	**Debug Clip Rectangle Maximum Y.** Sets the upper Y bound on pixels to be processed (if debug clip rectangle is enabled).
15:0	**Debug Clip Rectangle Maximum X.** Sets the upper X bound on pixels to be processed (if debug clip rectangle is enabled).

3.3.8.7 SF_RDATA—Debug Return Data

Address Offset:	7B18h–7B1Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**SF Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.3.9 Windower / Intermediate Z

The Windower Mask Unit (WM) will generate multiple dispatches per polygon. Each payload that is dispatched has subspan position information which is carried along with the payload. The Windower will generate an incrementing thread ID for each payload dispatched.

R0.7		
SS Flag	Reserved	**Primitive Sequence Number**
1 bit	7 bits	24 bits

R0.6	
Reserved for SW Debug	**Thread ID**
8 bits	24 bits

The thread number issued by the WM will be a primitive relative count. Two debug fields will be sent through the payload dispatch, a primitive sequence number and an incrementing thread ID. The WM will have the ability to snapshot a subspan of any input primitive according to the subspan's XY coordinates. Based on the snapshot, the WM will output to an MMIO register the thread ID that contained that subspan.

A snapshot can now be taken using the thread ID of the subspan of interest. This snapshot flag can be sent to the thread dispatcher.

A count field can be supplied via MMIO which will cause the WM snapshot to be generated only on the nth occurrence of that subspan. The WM will count how many times the selected subspan occurs. The subspan count can be read by setting **Snapshot Output Mux Select** to "Subspan Instance Count").

If the upstream snapshot flag is set, the Windower will only snapshot subspans that belong to the primitive identified in the SF snapshot compare.

The following pseudo code describes the Windower snapshot behavior.

```
if (Enable Subspan Instance Compare) {
  snapshot compare match = subspan x,y match && subspan instance
count match
} else {
  snapshot compare match = subspan x,y match
}

if (snapshot all threads) {
  local snapshot flag = 1
} else if (snapshot enable) {
  local snapshot flag = snapshot compare match
}

if (Use Upstream Snapshot) {
  final snapshot flag = upstream snapshot flag && local snapshot
flag
} else {
  final snapshot flag = local snapshot flag
}

if {Thread Snapshot Enable) [
  snapshot flag to payload = final snapshot flag
} else {
  snapshot flag to payload = 0
}
```

3.3.9.1 WIZ_CTL—Debug Control

Address Offset:	7C00h–7C03h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30:29	Reserved : MBZ
28:26	Reserved. MBZ
25	Reserved. MBZ
24	Reserved. MBZ
23:16	**Subspan Instance Number.** If **Subspan Instance Compare** is enabled, identifies which occurrence of a subspan to trap on. Format:U8 Range:1 – 255
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned 0 = Windower Kernel Pointer 1 = Subspan Instance Count 2 = Primitive Sequence Number from SF 3 – 255 = = Defined in a Windower-specific document outside the PRM
7	Reserved : MBZ
6	**Single Subspan Dispatch.** When set, WIZ will only dispatch one subspan per payload for non-promoted or computed IZ cases. For the A stepping of BW this bit should always be set to 1.
5	**Ignore Color Scoreboard Stalls.** When set, WIZ will ignore pixel dispatch scoreboard blocking conditions and continue to dispatch new subspans.
4	**Enable Subspan Instance Compare.** When set, the WIZ snapshot will trap on the nth occurrence of the selected subspan, where n is the value entered as the Subspan Instance Number above (bits 23:16). If this bit is clear the first occurrence of the selected subspan will be trapped
3	**Use Upstream Snapshot Flag.** This bit is set to enable trapping data in the Windower based on a snapshot flag passed downstream from the SF unit. If set, the downstream snapshot flag from SF *and* the subspan X, Y comparison must succeed in order for the current subspan to be trapped. If this bit is clear the snapshot flag passed down from SF is ignored and any matching subspan will be trapped (subject to instance number if **Enable Subspan Instance Compare** is set.).
2	**Snapshot All Threads.** If set, the snapshot flag will be set for all threads generated by this Fixed Function Unit (Overrides Snapshot Enable bit). If set, the data for the last thread executed will be captured in the FF unit (qualified in SVG with global debug enable).
1	**Thread Snapshot Enable.** This bit is set to enable passing the snapshot flag into the pixel shader thread dispatch. If Snapshot All Threads is also set, the snapshot flag will be passed into the thread dispatch for every thread (qualified in SVG with global debug enable).
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the WIZ (qualified in SVG with global debug enable).

3.3.9.2 WIZ_STRG_VAL —Debug Snapshot Trigger Value

Address Offset:	7C04h–7C07h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	**Pixel Y Compare Value.** This field controls which pixel Y value to trap on. This value must match the current subspan Y in order for it to be trapped (lsb of the compare value is ignored by hw).
15:0	**Pixel X Compare Value.** This field controls which pixel X value to trap on. This value must match the current subspan X in order for it to be trapped (lsb of the compare value is ignored by hw).

3.3.9.3 WIZ_RDATA—Debug Return Data

Address Offset:	7C08h–7C0Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**WIZ Debug Data.** Returns data captured by the compare on snapshot subspan x, y (based on snapshot output mux select).

3.3.10 Video Front End

VFE passes down a unique 32 bit debug identifier to TS with each kernel it generates. This 32-bit field will be passed into the thread as in DW6 of R0 header. Contents within the debug identifier as well as the mechanism to generate the field can be found in the *Media* chapter.

3.3.10.1 VFE_CTL—Debug Control

Address Offset:	7D00h–7D03h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After debugging the errant thread, the driver should reset this bit to 0.
30:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned 0 – 255 = = Defined in a VFE-specific document outside the PRM
7:1	Reserved : MBZ
0	**Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in the VFE (qualified in SVG with global debug enable).

3.3.10.2 VFE_STRG_VAL—Debug Snapshot Trigger Value

Address Offset:	7D04h–7D07h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:28	Reserved : MBZ
27:0	**Snapshot ID.** This field will be used by the FF logic to compare to the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. Various signals within the FF unit will also be latched for read back via MMIO registers.

3.3.10.3 VFE_RDATA—Debug Return Data

Address Offset:	7D08h–7D0Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**VFE Debug Data.** Returns data captured during a snaphot

3.3.11 Thread Spawner

TS does not have a kernel counter.

To assemble the r0 register when creating a root thread, TS simply copies the 32-bit debug identifier received from VFE into r0.6, and leaves 0 in r0.7.

For child thread, r0 register is created by the parent thread and stored in URB. TS is not involved. However, the dispatch child thread message sent to TS must contain the same debug field (r0.6 and r0.7) for the given child thread. TS latches the debug field.

TS can generate the snapshot based on snapshot match, or generate the snapshot independently for all root threads (including synchronized root) and/or for all child threads depending the configuration in Debug Register 1.

When Thread Snapshot Enable is set, but "Snapshot All Root Threads" and "Snapshot All Child Threads" are not set, TS uses Debug Register 2 and 3 to match against the thread to be dispatched. If it is a root thread, TS uses the debug field received from VFE. If it is a child thread, TS uses the debug field latched from the child dispatch request message.

3.3.11.1 TS_CTL—Debug Control

Address Offset:	7E00h–7E03h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After debugging the errant thread, the driver should reset this bit to 0.
30:16	Reserved : MBZ
15:8	Debug Data Mux Select: Controls which 32 bits of the trap data for the selected Thread Spawner unit are returned 0 = Message Error: [31:27] Message Sideband Function Control [4:0] [26] End of Thread [25:22] FFID[3:0] [21:18] EUID[3:0] [17:16] TID[1:0] [15] Error Code Valid [14:10] Dispatch ID[4:0] [9:1] URB Handle[8:0] [0] Error Code; Identifies Type of Error; 0 = Unexpected Message to TS; 1 = Bad Length 1 – 2 = Defined in a Thread Spawner-specific document outside the PRM 3 = Snapshot Interface Descriptor: [31:30] Reserved [29] Child Thread [28] Reserved [27:0] Interface Descriptor 4 – 255 = Defined in a Thread Spawner-specific document outside the PRM
7:3	Reserved : MBZ
2	**Snapshot All Child Threads.** If set, the snapshot flag will be set for all spawned threads generated by TS (qualified in SVG with global debug enable).
1	**Snapshot All Root Threads.** If set, the snapshot flag will be set for all root threads generated by TS (qualified in SVG with global debug enable).
0	**Thread Snapshot Enable.** This bit is set to enable the fixed function snapshot logic in TS and to pass the snapshot flag into TS thread dispatch (qualified in SVG with global debug enable).

3.3.11.2 TS_STRG_0-6VAL—Debug Snapshot Trigger R0.6 Value

Address Offset:	7E04h–7E07h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved : MBZ Note: Bits 31:24 of R0.6 are reserved. Therefore, this field is not included in the comparison.
23:0	**Snapshot ID0 (bits [23:0] of R0.6).** This field together with **Debug Snapshot Trigger R0.7 Value** will be used by the FF logic to compare to the debug field (R0.6 and R0.7) of the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. TS does not latch any internal signals upon a snapshot is generated.

3.3.11.3 TS_STRG_0-7VAL—Debug Snapshot Trigger R0.7 Value

Address Offset:	7E08h–7E0Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved : MBZ Note: Bit 31 of R0.7 carries the snapshot flag, and bits [30:24] of R0.7 are reserved. Therefore, this field is not included in the comparison.
23:0	**Snapshot ID1 (bits [23:0] of R0.7).** This field together with **Debug Snapshot Trigger R0.6 Value** will be used by the FF logic to compare to the debug field (R0.6 and R0.7) of the current thread being dispatched. When a match occurs, a snapshot flag will be generated and passed into the thread dispatch. TS does not latch any internal signals upon a snapshot is generated.

3.3.11.4 TS_RDATA—Debug Return Data

Address Offset:	7E0Ch–7E0Fh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**TS Debug Data.** Returns data captured when a 'bad' message is received (based on mux select in TS Debug Control)

3.3.11.5 Parent Thread Recommendations

Kernel Development Guideline

A parent thread, serving the purpose of a fixed function unit, should generate the 32-bit r0.7 debug field for its child threads.

As the MSB of r0.7 is reserved for hardware to insert the snapshot flag, software must make sure that it never alters bit 31. This is shown by the following pseudo code.

```
and (1) reg32   r0.7:ud 0x80000000:ud          // save bit 31

…                                 // update r0.7 with new child ID

or (1) r0.7:ud r0.7:ud reg32:ud                // restore bit 31
```

A root thread should forward its r0.6 to its child thread.

A branch parent thread should create a unique parent ID in r0.6 for its child threads. For example, it may copy its own r0.7 into its children's r0.6.

A leaf child thread should not alter the debug ID field.

The debug ID field of a root thread:

R0.7		R0.6		
SSflag	0	**FF Unit ID**	**Object ID**	**Root Thread ID**
1 bit	31 bits	4 bits	12 bits	16 bits

The debug ID field of a child thread:

R0.7		R0.6		
SSflag	**Child Thread ID**	**FF Unit ID**	**Object ID**	**Parent Thread ID**
1 bit	31 bits	4 bits	12 bits	16 bits

3.4 Shared Function Debug

3.4.1 Thread Dispatcher

3.4.1.1 TD_CTL—Debug Control

Address Offset:	8000h-8003h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:8	**Debug Data Mux Select.** Controls which 32 bits of trap data are returned. 0 – 255 = = Defined in a Thread Dispatcher-specific document outside the PRM
7	**External Halt on R0 Debug Match.** When set, causes an external halt exception to occur on the thread dispatch for which the comparison on R0.6 and R0.7 succeeds. Setting this bit forces the External Halt Exception bit to be set to true. This signal is qualified in SVG with global debug enable.
6	**Force External Halt.** When set, forces an external halt exception to occur on the next thread dispatch. Setting this bit forces the External Halt Exception bit to be set to true. This signal is qualified in SVG with global debug enable.
5	**Exception Mask Override.** When set, forces all exception masks to be over-ridden with the settings in TD Debug Register 2 (qualified in SVG with global debug enable).
4	**Force Thread Breakpoint Enable.** When set, enables breakpoints on all dispatched threads. When clear, thread debug may still be enabled on a snapshot basis. CR1.15 will be set for all threads when this bit is set, regardless of the state of bit 2 and regardless of whether a snapshot was associated with the thread at the FF. This signal is qualified in SVG with global debug enable.
3	Reserved : MBZ
2	**Breakpoint Enable.** Enables breakpoints to be honored in the dispatched thread. This bit must be set in order for CR1.15 to be set for a thread corresponding to FF data that triggered a snapshot, unless bit 4 is set. When bit 4 is set this bit is ignored (qualified in SVG with global debug enable).
1:0	Reserved : MBZ

3.4.1.2 TD_CTL2—Debug Control 2

Address Offset:	8004h-8007h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:29	Reserved : MBZ
28	**Illegal Opcode Exception Override.** When set (and the exception override bit, TD Debug Control, is set), forces the illegal opcode exception to be enabled, overriding the state control of this function.
27	Reserved : MBZ
26	**MaskStack Exception Override.** When set (and the exception override bit, TD Debug Control, is set), forces the MaskStack exception to be enabled, overriding the state control of this function.
25	**Software Exception Override.** When set (and the exception override bit, TD Debug Control, is set), forces the software exception to be enabled, overriding the state control of this function.
24	Reserved : MBZ
23:19	Reserved : MBZ
18:16	**Active Thread Limit.** When enabled by **Active Thread Limit Enable**, the TD will limit the number of active threads per execution unit to the value specified in this field.
15:13	Reserved : MBZ
12:10	**Reserved : MBZ**
9	Reserved : MBZ
8	**Active Thread Limit Enable.** Limits the number of active threads per execution unit.
7	**Thread Spawner Execution Mask Enable.** Limits which execution units are available for these threads to execute on
6	**WIZ Execution Mask Enable.** Limits which execution units are available for these threads to execute on
5	**SF Execution Mask Enable.** Limits which execution units are available for these threads to execute on
4	**Clipper Execution Mask Enable.** Limits which execution units are available for these threads to execute on
3	**GS Execution Mask Enable.** Limits which execution units are available for these threads to execute on
2	Reserved : MBZ
1	Reserved : MBZ
0	**VS Execution Mask Enable.** Limits which execution units are available for these threads to execute on

3.4.1.3 TD_VF_VS_EMSK—Debug VF/VS Execution Mask

Address Offset:	8008h-800Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**VS Execution Mask.** When enabled, forces all VS threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.1.4 TD_GS_EMSK—Debug GS Execution Mask

Address Offset:	800Ch-800Fh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**GS Execution Mask.** When enabled, forces all GS threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.1.5 TD_CLIP_EMSK—Debug Clipper Execution Mask

Address Offset:	8010h-8013h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**Clipper Execution Mask.** When enabled, forces all Clipper threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.1.6 TD_SF_EMSK—Debug SF Execution Mask

Address Offset:	8014h-8017h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**SF Execution Mask.** When enabled, forces all SF threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.1.7 TD_WIZ_EMSK — Debug WIZ Execution Mask

Address Offset:	8018h-801Bh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**WIZ Execution Mask.** When enabled, forces all WIZ threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.1.8 TD_0-6_EHTRG_VAL—Debug R0.6 External Halt Trigger Value

Address Offset:	801Ch-801Fh
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved: MBZ
23:0	**R0.6 Debug Compare Value.** Specifies the match to compare against the R0.6 debug header. A match on both the R0.7 and R0.6 headers (based on the mask bits) will cause the TD to generate an external halt exception if **External Halt on R0 Debug Match** is set.

3.4.1.9 TD_0-7_EHTRG_VAL—Debug R0.7 External Halt Trigger Value

Address Offset:	8020h-8023h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved: MBZ
23:0	**R0.7 Debug Compare Value.** Specifies the match to compare against the R0.7 debug header. A match on both the R0.7 and R0.6 headers (based on the mask bits) will cause the TD to generate an external halt exception if **External Halt on R0 Debug Match** is set.

3.4.1.10 TD_0-6_EHTRG_MSK—Debug R0.6 External Halt Trigger Mask

Address Offset:	8024h-8027h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved: MBZ
23:0	**R0.6 Debug Compare Mask.** Specifies the mask to use for the compare against the R0.6 debug header. A match on both the R0.7 and R0.6 headers (based on the mask bits) will cause the TD to generate an external halt exception if **External Halt on R0 Debug Match** is set.

3.4.1.11 TD_0-7_EHTRG_MSK—Debug R0.7 External Halt Trigger Mask

Address Offset:	8028h-802Ch
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved: MBZ
23:0	**R0.7 Debug Compare Mask.** Specifies the mask to use for the compare against the R0.7 debug header. A match on both the R0.7 and R0.6 headers (based on the mask bits) will cause the TD to generate an external halt exception if **External Halt on R0 Debug Match** is set.

3.4.1.12 TD_RDATA—Debug Return Data

Address Offset:	802Ch-802Fh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**TD Debug data.** Returns debug data (based on debug data mux select)

3.4.1.13 TD_TS_EMSK—Debug TS Execution Mask

Address Offset:	8030h-8033h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:0	**TS Execution Mask.** When enabled, forces all TS threads to execute on only those execution units with the corresponding mask bit set to 1.

3.4.2 Math Unit

3.4.2.1 MATH_CTL—Math Debug Control

Address Offset:	8100h–8103h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved: MBZ
17:16	**EM Unit Select:** Controls which EM returns the trap data associated with the Snapshot Output Mux Select Range: 0 - 1

Bit	Descriptions
15:8	**Debug Data Mux Select:** Controls which 32 bits of the trap data for the selected SF unit are returned 0 = Error Message Header:

[31:28]	Reserved
[27:24]	FFID[3:0]
[23:22]	Reserved
[21:20]	TID[1:0]
[19:16]	EUID[3:0]
[15:12]	rlen[3:0]
[11:8]	mlen[3:0]
[7:4]	opcode[3:0]
[3:0]	Error Code[3:0] -Identifies type of error [0] = bad length [1]= invalid opcode [2] = EOT bit detected [3] = Error bit detected In all cases above EM will discard message

1 = EM Debug Data

31:31	Input Data Valid
30:30	Input FIFO Full
29:26	Input Opcode
25:22	Input EUID
21:20	Thread ID
19:16	Msg Length
15:9	Destination register
8	**Sequencer to FPU** Valid & !Hold
7 6:3	**FPU to FPU** Valid **Counter**
2	**FPU to Assembler** Valid & !Hold
1	**Output of EM** Valid & Grant
0	**Reserved**

2 -255 = to be defined is an EM-specific document outside the PRM

Bit	Descriptions
7:2	Reserved : MBZ
1	**Non-Pipeline Mode Enable.** When enabled, forces the Math Unit to operate in a non-pipelined mode of operation (qualified in SVG with global debug enable).
0	Reserved : MBZ

3.4.2.2 MATH_RDATA—Math Debug Return Data

Address Offset:	8104h–8107h
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Math Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.4.3 Instruction / State Cache

3.4.3.1 ISC_CTL—Instruction / State Debug Control

Address Offset:	8200h–8203h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:0	Reserved : MBZ

3.4.4 Instruction L1 Cache

3.4.4.1 ISC_L1CA_CTR—Instruction L1 Cache Debug Control

Address Offset:	8280h–8283h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned 0 – 255 = = Defined in an ISC-specific document outside the PRM
7:0	Reserved : MBZ

3.4.4.2 ISC_L1CA_RDATA—Instruction L1 Cache Debug Return Data

Address Offset:	8284h–8287h
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Instruction L1 Cache Debug Data.** Returns data (based on output mux select)

3.4.4.3 ISC_L1CA_BP_ADR1—Instruction L1 Cache Breakpoint Address 1 Control

The Instruction L1 Cache Breakpoint Address Control Registers allow a breakpoint to be set on a given instruction address. These registers may be updated at any time.

Address Offset:	8288h–828Ch
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:4	**Breakpoint Address 1.** This field holds the 28 MSBs of the desired linear address of the context's memory space.
3:1	Reserved : MBZ
0	**BPIP 1 Enable.** Specifies whether this breakpoint is enabled or disabled. This field has no effect when Breakpoint Enable bit is unset. This field is initialized to 0 at reset. 0 = Breakpoint IP 1 disabled 1 = Breakpoint IP 1 enabled

3.4.4.4 ISC_L1CA_BP_ADR2—Instruction L1 Cache Breakpoint Address 2 Control

Address Offset:	8290h–8293h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:4	**Breakpoint Address 2.** This field holds the 28 MSBs of the desired linear address of the context's memory space.
3:1	Reserved : MBZ
0	**BPIP 2 Enable.** Specifies whether this breakpoint is enabled or disabled. This field has no effect when Breakpoint Enable bit is unset. This field is initialized to 0 at reset. 0 = Breakpoint IP 2 disabled 1 = Breakpoint IP 2 enabled

3.4.4.5 ISC_L1CA_BP_OPC1—Instruction L1 Cache Breakpoint Opcode 1 Control

Address Offset:	8294h–8297h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved : MBZ
23:16	**Breakpoint Opcode 1.** Specifies opcode to breakpoint against (if enabled). Any thread for which breakpoints are enabled will break on any instruction matching this field.
15:2	Reserved : MBZ
1	**Breakpoint Opcode 1 EOT Enable.** Specifies opcode to breakpoint against (if enabled). Any thread for which breakpoints are enabled will break on any instruction matching this field.
0	**Breakpoint Opcode 1 Enable.** Specifies opcode to breakpoint against (if enabled). Any thread for which breakpoints are enabled will break on any instruction matching this field.

3.4.4.6 ISC_L1CA_BP_OPC2—Instruction L1 Cache Breakpoint Opcode 2 Control

Address Offset:	8298h–829Ch
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:24	Reserved : MBZ
23:16	**Breakpoint Opcode 2.** Specifies opcode to breakpoint against (if enabled). Any thread for which breakpoints are enabled will break on any instruction matching this field.
15:1	Reserved : MBZ
0	**Breakpoint Opcode 2 Enable.** Specifies opcode to breakpoint against (if enabled). Any thread for which breakpoints are enabled will break on any instruction matching this field.

3.4.5 Message Arbiter

When the MASFHalt bit is set in the debug register, the Message Arb enters a ‘halt’ state and ceases to further arbitrate, thus no further grants are issued back to EUs. Any in-flight messages or any ‘grant’ which has been committed at the time of transition is continued to its conclusion. Any previously-posted requests from the EUs to the MA remain in the MA’s input/request queues, and the MA’s input queue logic remains operational, capable of accepting further EU requests until the associated input queue is full. The MA remains in the ‘halt’ state until the register’s bit field is reset by driver software.

A similar mechanism is defined for the Writeback Arb (WBarb) called ‘WBHalt’. In this case, the arbiter halts further issuance of grants to shared functions which have pending requests in the WBarb’s input queues. Any pending requests which have already been committed are allowed to continue to completion. Similar to MAHalt, the request logic of WBarb continues to operate normally while in the ‘halt’ state, accepting new requests until such time that the request queues become full. The WBarb remains in the ‘halt’ state until the register’s bit is reset by driver software.

A similar mechanism also exists for the RowInput Arb (RIarb). A single bit is defined to place all the RI arbiters (one exists for each row) into halt mode. When this bit is set, the RI arbiters halt further arbitration of Math return data and Writeback traffic. Any traffic already committed for transmission is allowed to complete, and the RIarb input logic continues to accept any new requests that may be made. Note that thread-dispatch traffic is defined as non-throttled, so it will not be halted. The RIarb remains in the ‘halt’ state until the register’s bit is reset by driver software.

3.4.5.1 MA_DEBUG_1—Message Arbiter Debug Control

Address Offset:	8300h–8303h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:5	Reserved : MBZ
4	**Writeback Arbiter Halt Mode (WBHalt).** Puts the WB arbiter into halt mode. WB arbitrates between the writebacks (return data) from all the shared functions except extended math which instantiated per row and whose writeback data goes directly to the RI arbiter. This signal is qualified in SVG with global debug enable. Format = Enable
3	**RowInput Arbiter Halt Mode (RIHalt).** Puts the RI arbiter into halt mode. One RI per row arbitrates between writebacks (shared function return data) and TD dispatches coming into the EUs. This bit places the RI arbiters of all rows into halt mode. However, RI arbiters will still grant TD dispatch requests even when in halt mode. This signal is qualified in SVG with global debug enable. Format = Enable
2	**Shared Function Arb Halt Mode (MASFHalt).** Puts the MASF arbiter into Halt mode. MASF arbitrates all request messages to shared functions from EUs. This signal is qualified in SVG with global debug enable. Format = Enable
1	**Non-Pipeline Mode Enable.** Forces the Message Arbiter to operate in a non-pipelined mode of operation. In this mode it will enqueue only 1 request to each shared function at a time. A new request will not be enqueued until the SF indicates its input queue is empty. Depending on the SF, this may or may not de-pipeline the SF to some extent. This signal is qualified in SVG with global debug enable. Format = Enable
0	Reserved: MBZ

3.4.6 Sampler

3.4.6.1 SAMPLER_CTL—Sampler Debug Control

Address Offset:	8400h–8403h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:8	**Debug Data Mux Select.** Controls which 32 bits of the trap data for the selected sampler unit are returned 0 = Error Message Header: [31:28] — Reserved [27:24] — FFID[3:0] [23] — Reserved [22:21] — TID[1:0] [20] — Reserved [19:16] — EUID[3:0] [15:0] — Error Code[15:0]: Identifies type of error are: 1 = bad length 2 = invalid EOT message 1 = Debug data from Message Header dword 6 2 = Debug data from Message Header dword 7 3 – 255 = Defined in a Sampler-specific document outside the PRM
7:3	**Sampler Unit Select.** Selects the unit to be observed in the debug data register 00000 – SIUnit 00001 – PLUnit 00010 – DGUnit 00011 – QCUnit 00100 – FTUnit 00101 – DMUnit 00110 – SCUnit 00111 – FLUnit 01000 – SOUnit
2	**Texture L1 Cache Disable.** Disables the Texture L1 Cache. Unless this bit is set when the texture cache is guaranteed to be empty (such as during initialization), the texture cache must first be flushed by issuing an MI_FLUSH with the **Map Cache Invalidate** bit set. Format = Disable
1	Reserved : MBZ
0	Reserved : MBZ

3.4.6.2 SAMPLER_RDATA—Sampler Debug Return Data

Address Offset:	8404h–8407h
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Sampler Debug Data.** Returns debug data (based on snapshot output mux select)

3.4.7 Data Port

The Dataport captures message header information for the first detected error. Error detection can occur at the message arbiter or be internally detected by the dataport.

3.4.7.1 DP_CTL—Data Port Debug Control

Address Offset:	8500h–8503h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ

Bit	Descriptions
15:8	**Debug Data Mux Select.** Controls which 32 bits of the trap data are returned 0 = Error Message Header: [31:28] Reserved [27:24] FFID[3:0] [23] Reserved [22:21] TID[1:0] [20] Reserved [19:16] EUID[3:0] [15:14] Reserved [13] Error Bit: Wrong Return Length [12] Error Bit: Illegal Surface Format [10] Error Bit: Wrong Message Programming for Stateless Boundary [9] Error Bit: Wrong Message Programming for illegal Stateless Mode Set [8] Error Bit: Wrong Message Programming for Message Length [7] Error Bit: Illegal Address Alignment [6] Error Bit: Wrong Message Programming for Block Size [5] Error Bit: Illegal Surface Type [4] Error Bit: Illegal Target Cache [3] Error Bit: Error Bit Received from Message Sideband [2] Error Bit Invalid S/F Id [1] Error Bit Invalid EOT Message [0] Error Bit Bad Length 1 = Debug data from Message Header dword 7 2 = Debug data from Message Header dword 6 3 – 31 = Defined in a Dataport-specific document outside the PRM 32 – 63 = Binding Table Pointers 64 – 255 = Defined in a Dataport-specific document outside the PRM
7:0	Reserved : MBZ

3.4.7.2 DP_RDATA—Data Port Debug Return Data

Address Offset:	8504h–8507h
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**Data Port Debug Data.** Returns debug data (based on snapshot output mux select)

3.4.8 Render Cache

3.4.8.1 RC_CTL—RC Debug Control

Address Offset:	8600h–8603h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:16	Reserved : MBZ
15:8	**Snapshot Output Mux Select.** Controls which 32 bits of the trap data are returned 0 – 255 = = Defined in a Render Cache-specific document outside the PRM
7:3	Reserved : MBZ
2	**Default Color Enable.** Forces the Render Cache to output a constant color for all channels when enabled (only for pixels whose r0 header has the snapshot flag set). This signal is qualified in SVG with global debug enable.
1	**Non-Pipeline Mode Enable.** Forces the Render Cache to operate in a non-pipelined mode of operation. This signal is qualified in SVG with global debug enable.
0	**Snapshot Enable.** This bit is set to enable the snapshot logic in the RC. The snapshot logic is also dependent on the snapshot flag being set in the R0 header. This signal is qualified in SVG with global debug enable.

3.4.8.2 RC_DEF_CLR—RC Debug Force Default Color

Address Offset:	8603h–8607h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:0	**Default Color.** If Default Color is enabled, the Render Cache will output the default color for each channel (RGBA). The default color is assumed to be in the format of the render target (and aligned to the MSB).

3.4.8.3 RC_RDATA—RC Debug Return Data

Address Offset:	8608h–860Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**RC Debug Data.** Returns data captured by the compare on snapshot ID (based on snapshot output mux select)

3.4.9 Unified Return Buffer (URB)

The ability to trap a write to the URB (from a thread) will be provided. When the URB snapshot enable bit is set (URB Debug Control), the URB will trap any message that has the snapshot flag set in the R0 header. The n^{th} data phase will be trapped (where n is specified using the Snapshot Output Register control in the URB Debug Control).

3.4.9.1 URB_CTL—URB Debug Control

Address Offset:	8700h–8703h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31	**Snapshot Complete.** This bit will be set to 1 by hardware when a snapshot compare occurs. After reading the desired snapshot return values, the driver should reset this bit to 0.
30:20	Reserved : MBZ
19:16	**Snapshot Output Register Select.** Controls which register of the URB write message is trapped. Range: 0 – 15

Bit	Descriptions
15:8	**Debug Data Mux Select.** Controls which 32 bits of the trap data are returned. Controls which 32 bits of the trap data for the selected sampler unit are returned 0 = Error Message Header: [31:28] — Reserved [27:24] — FFID[3:0] [23] — Reserved [22:21] — TID[1:0] [20] — Reserved [19:16] — EUID[3:0] [15:0] — Error Code[15:0] Identifies type of error 4 = Invalid S/F Id 1 = Debug data from Message Header dword 6 2 = Debug data from Message Header dword 7 3 – 7 = Reserved for HW usage 8 = trapped message data bits 0 – 31 9 = trapped message data bits 32 – 63 10 = trapped message data bits 64 – 95 11 = trapped message data bits 96 – 127 12 = trapped message data bits 128 – 159 13 = trapped message data bits 160 – 191 14 = trapped message data bits 192 – 223 15 = trapped message data bits 224 – 255 16 – 255 = = Defined in a URB-specific document outside the PRM
7:1	Reserved : MBZ
0	**Snapshot Enable.** This bit is set to enable the snapshot logic in the URB. The snapshot logic is also dependent on the snapshot flag being set in the R0 header. This signal is qualified in SVG with global debug enable.

3.4.9.2 URB_RDATA—URB Debug Return Data

Address Offset:	8708h–870Bh
Default Value:	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**URB Debug Data.** Returns data captured by the compare on snapshot enable (based on snapshot output mux select)

3.4.10 Thread Spawner (TS)

TS is not only a fixed function to generate thread for the media pipeline, it is also a shared function allowing a thread running on EU to directly send message to. Therefore, it must handle 'bad' messages in a similar manner as other shared functions.

See the Fixed Function Debug section above for the debug registers defined for handling bad messages.

3.5 Attention Signaling from EU to Host

- [mkd: Text duplicated in Exceptions chapter – need to resolve]
- Each thread has dedicated signaling mechanism to/from the host, based on a 3-wire protocol (per thread) and the 'wait n1' instruction.
- Likely uses: (a) Debug breakpoint "suspend/resume" signaling (b) Software manipulated serial data stream (bit-banged) to the host, for use in debug, in the case where thread state save/restore via the dataport is unavailable/hung.
- Attention Signaling
  - o The following signals are provided per thread:
    - Attn – a 1b signal sent from the thread to the host, indicating that the associated thread desires attention. This signals is set to '1' via the 'wait n1' instruction and held set until an 'AClear' signal is received from the host, at which time the signal is reset to '0'.
    - AData – a 1b data signal sent from the thread to the host, indicating any data the thread wishes to communicate; Reflects the value of the architectural register CR0, bit 2, as set by the thread.
    - AClear – a 1b signal, send from the host to the thread, which resets the thread's internal Host-To-Thread-Notification bit (register N1, bit 0), as well as the Host-Notification-Data register (register CR0, bit 2). MMIO registers available to the host
    - Attn[63:0] – reflects the combined 64 'Attn' bits of the thread. Read-only.
    - AData[63:0] – reflects the combined 64 'AData' bits of the thread. Read-Only.
    - AClear[63:0] – the port through which the host issues a 'Clear' signal to one or more threads. Write-only; a write to this register causes a 1-clock pulse to be sent to the associated 'Clear' signal of any bit position set as a '1'. Signaling attention to the driver
    - All 64 'Attn' bits are 'OR'ed together to crate a 1b ATTN interrupt signal to the driver

3.5.1 EU_CTL—EU Debug Control

Address Offset:	8800h-8803h
Default Value:	00000000h
Access:	Read/Write
Size:	32 bits

Bit	Descriptions
31:19	Reserved : MBZ
18:16	**EU Select.** Controls which EU returns the debug data associated with the Debug Data Mux Select Range: 0 – 7
15:8	**Debug Data Mux Select.** Controls which 32 bits of the trap data are returned 0 – 255 = = Defined in an EU-specific document outside the PRM
7:0	Reserved : MBZ

3.5.2 EU_ATT—EU Debug Attention

Address Offset:	8810h-881Fh
Default Value:	00000000000000000000000000000000h
Access:	Read-Only
Size:	128 bits

Bit	Descriptions
127:32	Reserved : MBZ.
31:0	**Attention.** Reflects the Attention bits (msb is EU 0 thread 0).

3.5.3 EU_ATT_DATA—EU Debug Attention Data

Address Offset:	8820h-882Fh
Default Value:	00000000000000000000000000000000h
Access:	Read-Only
Size:	128 bits

Bit	Descriptions
127:32	Reserved : MBZ.
63:0	**Attention Data.** A 1b data signal sent from the thread to the host, indicating any data the thread wishes to communicate (msb is EU 0 thread 0).

3.5.4 EU_ATT_CLR—EU Debug Attention Clear

Address Offset:	8830h-883Fh
Default Value:	00000000000000000000000000000000h
Access:	Write-Only
Size:	128 bits

Bit	Descriptions
127:32	Reserved : MBZ.
31:0	Attention Clear. A 1b data signal sent from the host to a thread, indicating the thread may resume operation (lsb is EU 0 thread 0). These bits must be set to 1, then set back to 0 to clear the associated thread wait.

3.5.5 EU_RDATA—EU Debug Return Data

Address Offset:	8840h–8843h
Default Value	UUUU UUUUh
Access:	Read-Only
Size:	32 bits

Bit	Descriptions
31:0	**EU Debug Data.** Returns debug data (based on debug data mux select)

3.6 Breakpoints

A breakpoint is an instruction attribute that will cause a breakpoint exception to be taken prior to issuing the instruction. A breakpoint is indicated for a given instruction in any one or more of these 3 ways:

- Setting the DebugCtrl field to '1' (**Breakpoint**) in the instruction word in memory. See *Instruction Set Summary* for details.
- Setting the **Breakpoint Address** field in one of the Instruction L1 Cache Breakpoint Address Control registers to the address of the instruction (See Instruction L1 Cache earlier in this chapter).
- Setting of the **Breakpoint Opcode** field in one of the Instruction L1 Cache Breakpoint Opcode Control registers to the opcode of the instruction. Note that any instructions with matching opcodes will raise a breakpoint exception if breakpoints are enabled.

In order for a breakpoint exception to be raised for these cases, the thread executing the instruction for which a breakpoint is indicated must have breakpoints enabled. This is controlled via the **Breakpoint Enable** bit in control register 1 (CR1). This control bit will be set when a thread is dispatched if:

- The **Breakpoint Enable** bit is set in the TD Debug Control registers (see Section TD_CTL—Debug Control) *and* the FF initiating the thread set the Snapshot Flag (see Section 3.2, The Snapshot Mechanism), *or*
- The **Force Thread Breakpoint Enable** bit is set in the TD Debug Control registers.

Otherwise, breakpoints will not be enabled for the thread by default. It is expected that many thread instantiations will be running the same instruction set from the same place in memory; the mechanism in the first bullet above makes it possible to take a breakpoint exception only in the thread that generated erroneous data (as detected by the Snapshot mechanism in the FF unit) instead of requiring inspection and manual restart of every thread instance running the same code until the instance of interest is found.

Setting of the **Breakpoint Suppress** bit in a thread's control register 0 (CR0) will prevent a breakpoint exception from being raised. This bit is cleared by hardware upon suppression of a single breakpoint; its purpose is to allow execution of an instruction with a breakpoint set to continue after the breakpoint exception handling is complete.

3.6.1 Single Stepping

An important method for debugging is single-stepping through code, allowing registers and memory to be examined after each instruction to make sure all intermediate results are as expected. Single-stepping is accomplished using the breakpoint mechanism.

A breakpoint is set (using any of the mechanisms above) at the point in the code where the debugger wants to begin single-stepping. The System Routine (SR) code that handles the breakpoint exception will normally communicate with host software where the majority of the debugger code resides. When the user wants to execute the next instruction, the SR sets the **Breakpoint Suppress** bit of CR0 but does not clear the **Breakpoint Exception Status and Control** bit of CR1. Control is transferred from the system routine back to application code by writing a 0 to the **Master Exception State and Control.**

Breakpoint Suppress will allow one instruction to execute before it is automatically reset by hardware, at which time another breakpoint exception will be taken since the **Breakpoint Exception Status and Control** bit is still set. In this way the debugger can step through the code executing just one instruction at a time. To end single-stepping, the SR simply clears the **Breakpoint Exception Status and Control** bit (the Breakpoint Suppress bit must still be set) prior to transferring control back to the application code. No more breakpoint exceptions will occur unless another breakpoint is encountered in the code or via matching one of the L1 Cache Breakpoint Control values described above.

3.6.2 Modification of Instruction Stream

A mechanism is provided to flush all caches in the path between main memory and the EUs. This is controlled via an MMIO bit (see the Memory Interface Registers chapter). This allows dynamic modification of the thread's code at any breakpoint. Note that the EUs prefetch instructions internally; these will be flushed on the branch to SIP and again on the return back to AIP.

3.7 Message Errors

3.7.1.1 Error-Types Visible to the Arb

- Case 1: Unknown SFID destination in 'send' inst
  - Message arbiter treats as a normal message.
  - URB decodes invalid SFID.
- Case 2: Message length too long for destination S/F input buffer
  - Detected by Msg Arb
  - Arb overrides message length for arbitration purposes only (sets length =1)
  - Message participates in arb as normal (w/ overridden shorter length)
  - Grant issued for the full message to the EU, w/ ERR signal supplied to EU
  - EU passes along ERR signal on the sideband w/ the message.
  - The S/F treats this as a bad message (see case 1 in the subsequent section).

3.7.1.2 Non-pipelined S/F Operation

- There may be cases of errors which are caused by the contents of the data payload (e.g. bad pointer, bad operand, etc.). These error types may not be apparent until the message is well down the S/F's pipeline. Given that many other messages may be in-flight in the pipeline also, it may be difficult to associate a failure to a specific message.
- To allow for message-to-error association, each S/F should implement a "non-pipelined" operational mode.
- Arbiter operation in Non-pipelined mode.
    - The arbiter guarantees that it will allow only one outstanding message to each S/F at a time. Normally the arbiter tracks the input buffer depths of all shared functions, and if sufficient room exists, allows a pending message to be sent. In Non-pipelined mode, the arbiter treats each input queue are either full or empty - if the input buffer is not empty, it is considered full and thus unable to receive a further message.
- Shared function operation in non-pipelined mode
    - The S/F preserves the message sideband and r0 data phase of all messages it receives into its Error register. The 'Valid' is not set at this time, thus no error condition signaled.
    - The S/F processes the message as normal.
    - If any error occurs during the processing of the message, the 'Valid' bit is set in the Error register and the message discarded. The S/F must continue to operate on all subsequent messages.
    - The S/F assists the message arb in ensuring that only 1 message is delivered at a time by not releasing its input buffer space until the current message is guaranteed to complete w/o error.

§§

4 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 Gen4 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 Gen4 Core in order to complete the *sample* instruction.

Subfunction	Description
Texture Coordinate Processing	Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include: cube map intersection.
Texel Address Generation	The Sampling Engine 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 miplevel weighting factors to be used in the subsequent filtering operations.
Texel Fetch	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.
Texture Palette Lookup	For streams which have "paletted" texture surface formats, this function uses the "index" values read from the texture map to look up texel color data from the texture palette.
Shadow Pre-Filter Compare	For shadow mapping, the texel samples are first compared to the 3rd (R) component of the pixel's texture coordinate. The boolean results are used in the texture filter.
Texel Filtering	Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This "combination" ranges from simply passing through a "nearest" sample to blending the results of anisotropic filters performed on two mipmap levels. The output of this function is a single 4-component texel value.
Texel Color Gamma Linearization	Performs optional gamma decorrection on texel RGB (not A) values.

4.1 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.

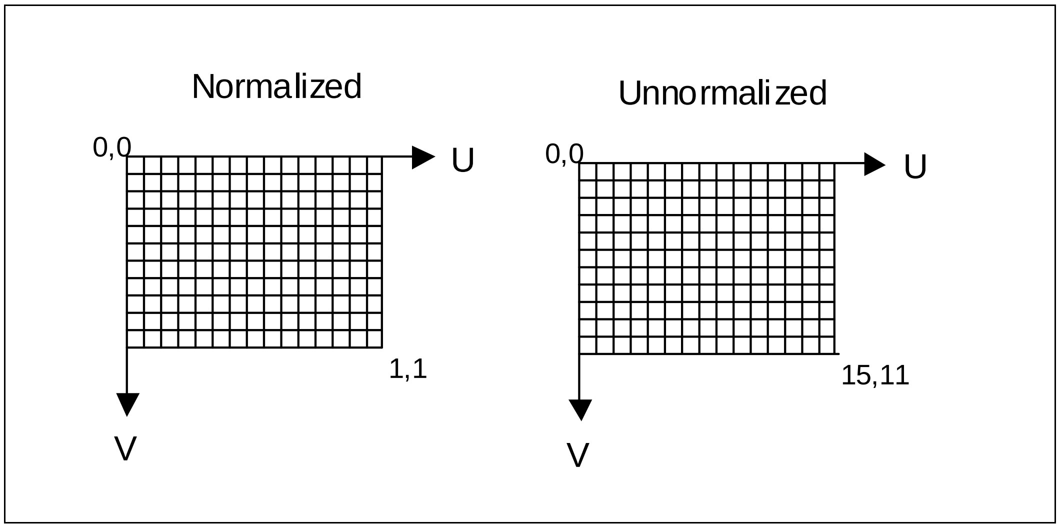
4.1.1 Texture Coordinate Normalization

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

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

Unnormalized coordinates are in units of texels and have not been divided (normalized) by the associated map's height or width. Here the origin is the located at the upper/left edge of the upper left texel of the base texture map. Unnormalized coordinates delivered to the sampling engine are only supported with the "ld" type messages.

Figure 4-1. Normalized vs. Unnormalized Texture Coordinates



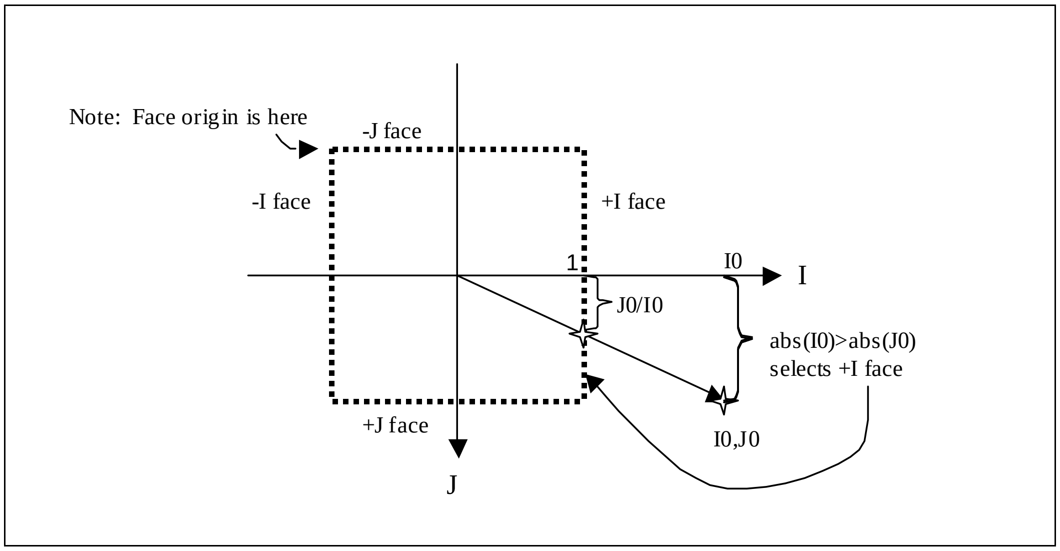
4.1.2 Texture Coordinate Computation

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:

Figure 4-2. Cube Map Coordinate Computation Example



4.2 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.

4.2.1 Level of Detail Computation (Mipmapping)

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

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

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

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

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

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

4.2.1.1 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.

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

$$\text{where:}$$

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

4.2.1.2 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.

4.2.1.3 LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable.

After biasing and/or adjusting of the LOD , the computed LOD value is clamped to a range specified by the (integer and fractional bits of) *MinLOD* and *MaxLOD* state variables prior to use in Min/Mag Determination.

MaxLOD specifies the lowest resolution mip level (maximum LOD value) that can be accessed, even when lower resolution maps may be available. Note that this is the only parameter used to specify the number of valid mip levels that be can be accessed, i.e., there is no explicit "number of levels stored in memory" parameter associated with a mip-mapped texture. All mip levels from the base mip level map through the level specified by the integer bits of *MaxLOD* must be stored in memory, or operation is UNDEFINED.

MinLOD specifies the highest resolution mip level (minimum LOD value) that can be accessed, where LOD==0 corresponds to the base map. This value is primarily used to deny access to high-resolution mip levels that have been evicted from memory when memory availability is low.

MinLOD and *MaxLOD* have both integer and fractional bits. The fractional parts will limit the inter-level filter weighting of the highest or lowest (respectively) resolution map. For example if *MinLOD* is 4.5 and *MipFilter* is LINEAR, LOD 4 can contribute only up to 50% of the final texel color.

4.2.1.4 Min / Mag Determination

The biased and clamped LOD is used to determine whether the texture is being minified (scaled down) or magnified (scaled up).

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

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

If the biased LOD is positive, the texture is being minified. In this case the *MipFilter* state variable specifies whether one or two mip levels are to be included in the texture filtering, and how that (or those) levels are to be determined as a function of the computed LOD.

4.2.1.5 LOD Computation Pseudocode

This section illustrates the LOD biasing and clamping computation in pseudocode, encompassing the steps described in the previous sections. The computation of the initial per-pixel LOD value *LOD* is not shown.

```
Bias:       S4.4
MinLod:     U4.6
MaxLod:     U4.6
Base:       U4.1
MIPCnt:     U4
SurfMinLod: U4

MaxLod = min(MaxLod, MIPCnt) + SurfMinLod
MinLod = min(MinLod, MIPCnt) + SurfMinLod

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

If (PreClamp)
    LOD = min(LOD, MaxLod)
    LOD = max(LOD, MinLod)

MagMode = (LOD - Base <= 0)
If (MagMode or MipFlt = None)
    LOD = 0
        LOD = min(LOD, ceil(MaxLod))
    LOD = max(LOD, floor(MinLod))
else if (MipFlt = Nearest)
    LOD += 0.5
    LOD = min(LOD, ceil(MaxLod))
        LOD = max(LOD, floor(MinLod))
        LOD = floor(LOD)
else  // MipFlt = Linear
        LOD = min(LOD, MaxLod)
    LOD = max(LOD, MinLod)
    TriBeta = frac(LOD)
    $LOD_0$ = floor(LOD)
    $LOD_1$ = $LOD_0$ + 1
```

4.2.2 Inter-Level Filtering Setup

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

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

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

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

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

4.2.3 Intra-Level Filtering Setup

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

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

The following table summarizes the intra-level filtering modes.

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

4.2.3.1 MAPFILTER_NEAREST

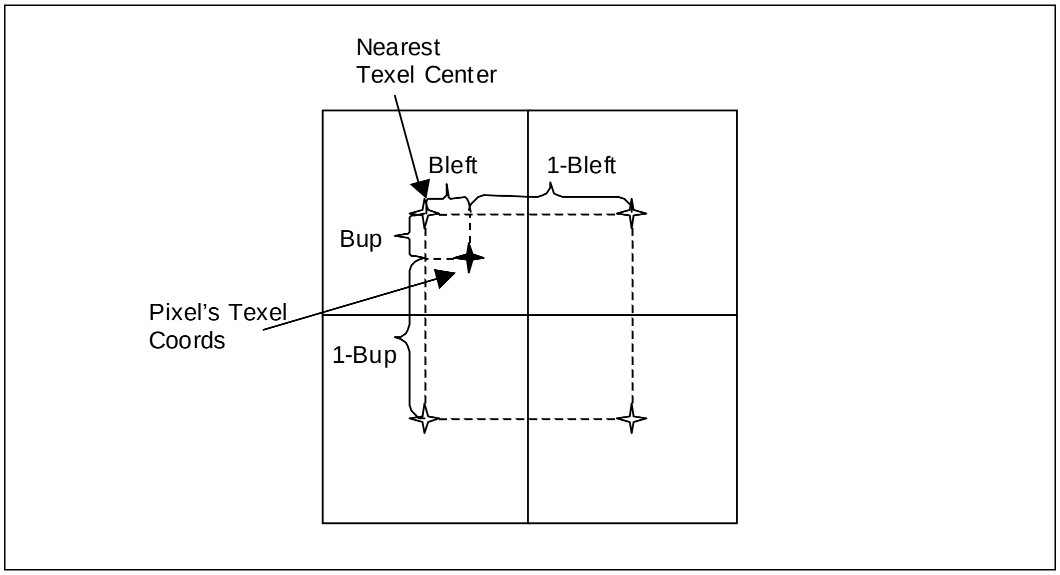
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level.

4.2.3.2 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.

Figure 4-3. Bilinear Filter Sampling

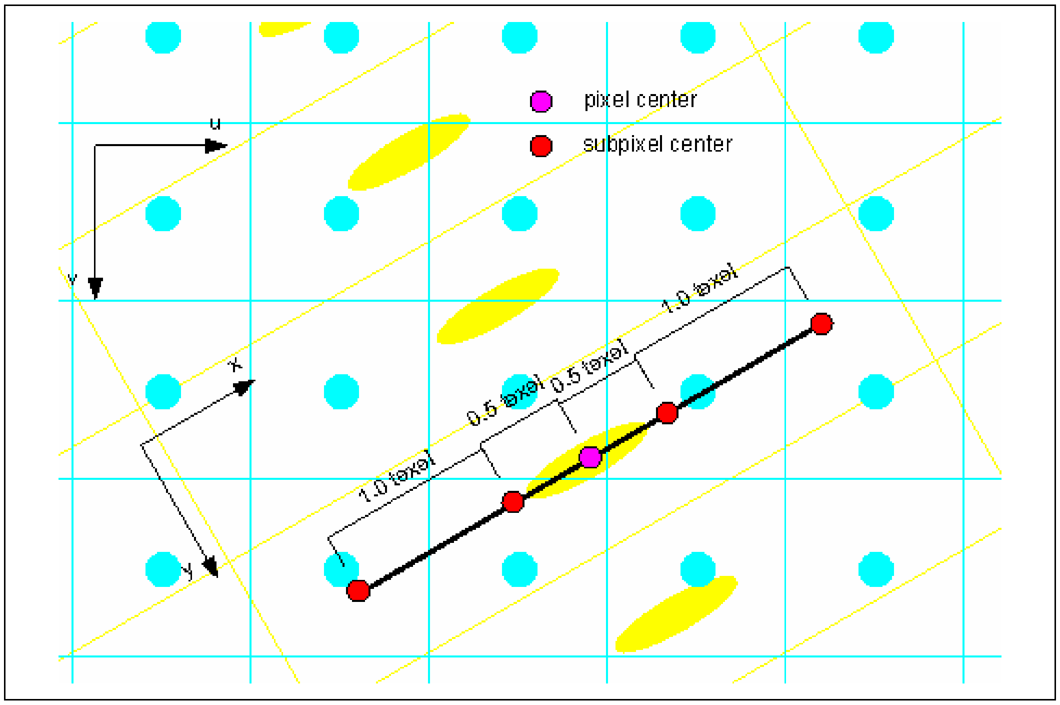


The four texels surrounding the pixel center are chosen for the bilinear filter. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

4.2.3.3 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.

4.2.3.4 MAPFILTER_MONO

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

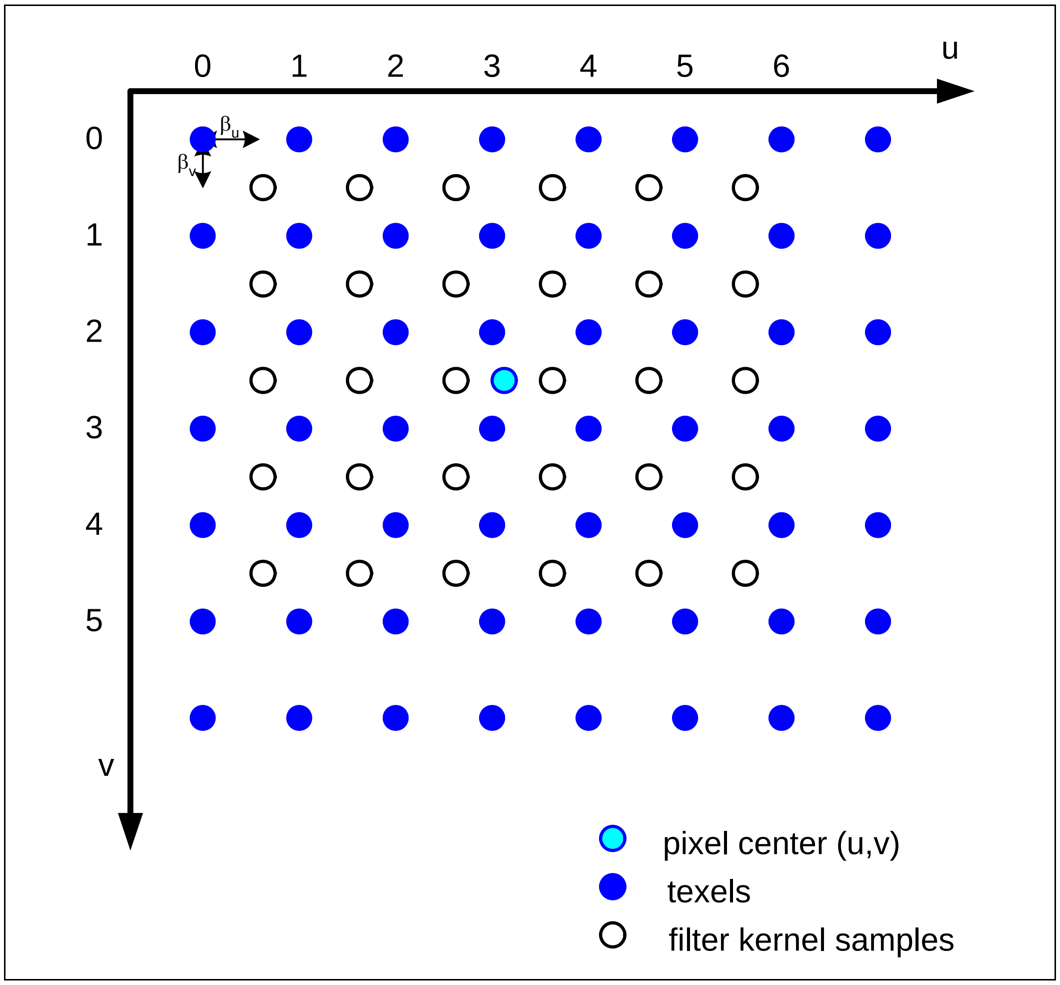
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f, v_f) relative to the pixel center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

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

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

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

Figure 4-4. Sampling Using MAPFILTER_MONO



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

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

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

4.2.4 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 <u>outside</u> 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 will need to specify TEXCOORDMODE_WRAP mode for the sampler that is provided with wrap-shortest texture coordinates, or artifacts may be generated along map edges.

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

Separate controls are provided for texture TCX, TCY, TCZ coordinate components so, for example, the TCX coordinate can be wrapped while the TCY coordinate is clamped. Note that there are no controls provided for the TCW component as it is only used to scale the other 3 components before addressing modes are applied.

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).

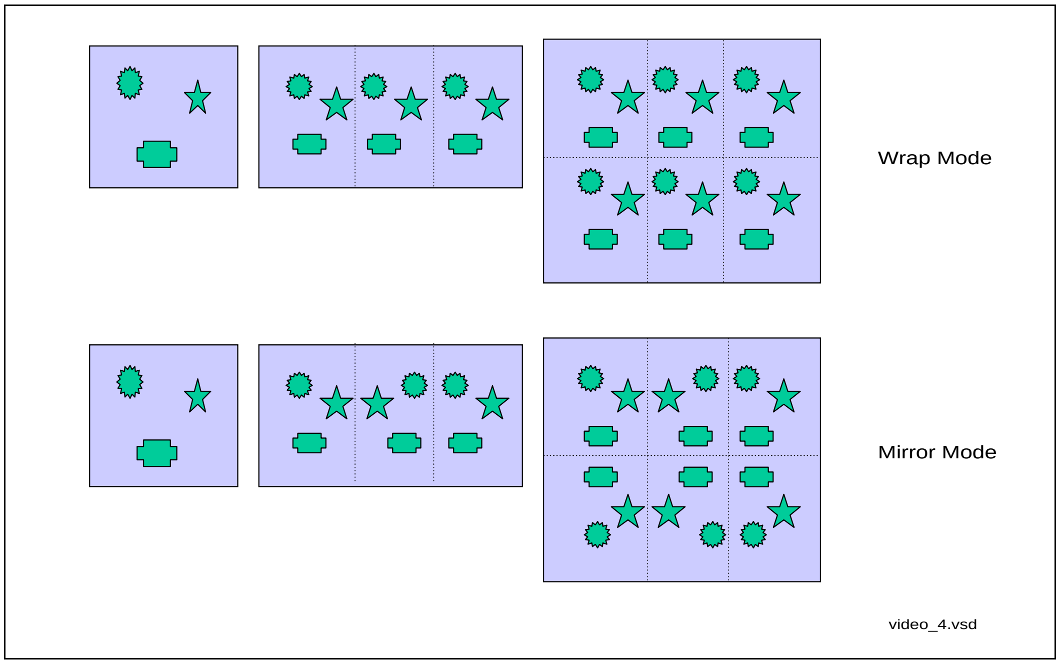
4.2.4.1 TEXCOORDMODE_WRAP Mode

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).

4.2.4.2 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.

Figure 4-5. Texture Wrap vs. Mirror Addressing Mode



4.2.4.3 TEXCOORDMODE_MIRROR_ONCE Mode

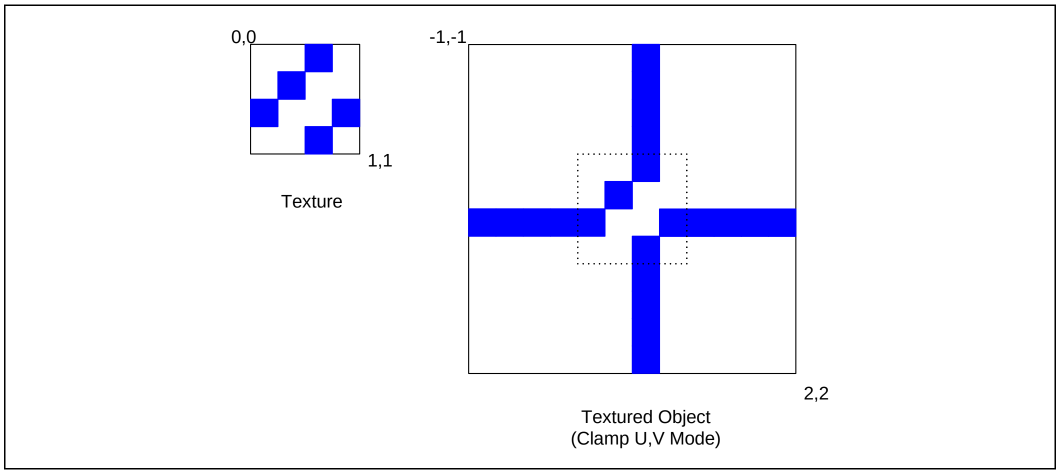
The TEXCOORDMODE_MIRROR_ONCE addressing mode is a combination of Mirror and Clamp modes. The absolute value of the texture coordinate component is first taken (thus mirroring about 0), and then the result is clamped to 1.0. The map is therefore mirrored once about the origin, and then clamped thereafter. This mode is used to reduce the storage required for symmetric maps.

4.2.4.4 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.

Figure 4-6. Texture Clamp Mode



4.2.4.5 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.

4.2.4.6 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.

4.3 Texel Fetch

The Texel Fetch function of the Sampling Engine reads the texture map contents specified by the texture addresses associated with each texel sample. The texture data is read either directly from the memory-resident texture map, or from internal texture caches. The texture caches can be invalidated by the **Sampler Cache Invalidate** field of the MI_FLUSH instruction or via the **Read Cache Flush Enable** bit of PIPE_CONTROL. Except for consideration of coherency with CPU writes to textures and rendered textures, the texture cache does not affect the functional operation of the Sampling Engine pipeline.

When the surface format of a texture is defined as being a compressed surface, the Sampler will automatically decompress from the stored format into the appropriate [A]RGB values. The compressed texture storage formats and decompression algorithms can be found in the *Memory Data Formats* chapter. When the surface format of a texture is defined as being an index into the texture palette (format names includiong "Px"), the palette lookup of the index determines the appropriate RGB values.

4.3.1 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.

4.3.1.1 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.

4.3.1.2 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.

4.4 Shadow Prefilter Compare

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

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

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

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

Programming Notes:

- Refer to the Surface Formats table in section 4.7.2.1.1 for the specific surface formats that are supported with shadow mapping.

4.5 Texel Filtering

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

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

4.6 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 post-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.

4.7 State

4.7.1 BINDING_TABLE_STATE

The binding table binds surfaces to logical resource indices used by shaders and other compute engine kernels. It is stored as an array of up to 256 elements, each of which contains one dword as defined here. The start of each element is spaced one dword apart. The first element of the binding table is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31:5	**Surface State Pointer.** This 32-byte aligned address points to a surface state block. This pointer is relative to the **Surface State Base Address**. **[DevBW-A,B] Errata BWT007:** Surface State data pointed at by offsets from Surface State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ

4.7.2 SURFACE_STATE

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

Surface state defines the state needed for the following objects:

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

4.7.2.1 For Most Messages

SURFACE_STATE		
Project: All		
This is the normal surface state used by all messages that use SURFACE_STATE except *deinterlace* and *sample_8x8*.		
DWord	**Bit**	**Description**
0	31:29	**Surface Type** Project: All Format: U3 enumerated type FormatDesc This field defines the type of the surface.
	28	**Reserved** Project: All Format: MBZ

Value	Name	Description	Project
0h	SURFTYPE_1D	Defines a 1-dimensional map or array of maps	All
1h	SURFTYPE_2D	Defines a 2-dimensional map or array of maps	All
2h	SURFTYPE_3D	Defines a 3-dimensional (volumetric) map	All
3h	SURFTYPE_CUBE	Defines a cube map or array of cube maps	All
4h	SURFTYPE_BUFFER	Defines an element in a buffer	All
5h-6h	Reserved		All
7h	SURFTYPE_NULL	Defines a null surface	All

Programming Notes
A null surface will be used in instances where an actual surface is not bound. When a write message is generated to a null surface, no actual surface is written to. When a read message (including any sampling engine message) is generated to a null surface, the result is all zeros. All of the remaining fields in surface state are ignored for null surfaces, with the following exceptions: • **Width**, **Height**, **Depth**, **LOD**, **MIP Map Layout Mode**, and **Render Target View Extent** fields must match the depth buffer's corresponding state for all render target surfaces, including null.
The **Surface Type** of a surface used as a render target (accessed via the Data Port's Render Target Write message) must be the same as the **Surface Type** of all other render targets and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless either the depth buffer or render targets are SURFTYPE_NULL.

SURFACE_STATE

27

Data Return Format

Project:	All	
Format:	U1 enumerated type	FormatDesc

For Sampling Engine Surfaces:

This field determines the format of the return data from the sampling engine to the compute engine. This field is ignored for surfaces used by other units.

For Other Surfaces:

This field is ignored.

Value	Name	Description	Project
0h	DATA_RETURN_FLOAT32	FLOAT32 data is returned	All
1h	DATA_RETURN_S1.14	S1.14 fixed point data is returned	All

Programming Notes
The S1.14 return format is only legal for returning data from normalized (UNORM, or SNORM) map formats where *all* channels have <= 8 bits. *It is not legal to use this format with any floating point or integer map format.*
S1.14 return format is only used for SIMD16 and SIMD8 messages from the sampling engine. For SIMD4x2 messages, FLOAT32 format will be used for surfaces specifying S1.14 data return format.
Data returned in format S1.14 will be converted to FLOAT32 before reaching the GRF register, thus the state of this bit does not affect the kernel.
It is recommended that S1.14 format be used wherever it is legal, as the performance will generally be improved.

26:18

Surface Format

Project:	All	
Format:	U32	FormatDesc

Specifies the format of the surface or element within this surface. This field is ignored for all data port messages other than the render target message. Some forms of the media block messages use the surface format.

Refer to the table in section 4.7.2.1.1 for the formats supported and their encodings.

Programming Notes
Tile Walk TILEWALK_YMAJOR is UNDEFINED for *render target* formats that have 128 bits-per-element (BPE).
YUV (YCRCB) surfaces used as render targets can only be rendered to using 3DPRIM_RECTLIST with even X coordinates on all of its vertices, and the pixel shader cannot kill pixels.

Errata	Description	Project
#	surfaces with FLOAT format are not supported.	[DevBW-A,B]

SURFACE_STATE

	17:14	**Color Buffer Component Write Disables** Project: All Format: U4 bit mask of disables (0 or logical OR of any of the enumerated values) FormatDesc **For Render Target Surfaces:** This field contains a bitmask that controls the writing of individual color components into the Color Buffer. If a component is disabled (bit set) writes to the color buffer will not modify that component. If enabled (bit clear), that component can be overwritten. **For Other Surfaces:** this field is ignored.

Value	Name	Description	Project
1000b	WRITEDISABLE_ALPHA		All
0100b	WRITEDISABLE_RED		All
0010b	WRITEDISABLE_GREEN		All
0001b	WRITEDISABLE_BLUE		All

Programming Notes
For YUV surfaces, this field must be set to 0000B (all channels enabled).

Errata	Description	Project
#	Desc	All

	13	**Color Blend Enable** Project: All Format: Enable FormatDesc **For Render Target Surfaces:** Specifies that color blend is enabled for this particular render target. The Color Buffer Blend Enable state in COLOR_CALC_STATE provides global control over blending. See Color Buffer Blending (Windower) for details. **For Other Surfaces:** is ignored.

Errata	Description	Project
#	This **Color Blend Enable** bit is not used, and acts as if it is ENABLED for each RenderTarget. Blending is enabled or disabled only a a global basis by the **Color Buffer Blend Enable** state variable in COLOR_CALC_STATE.	[DevBW-A,B]

SURFACE_STATE		
	12	**Vertical Line Stride** Project: All Format: U1 in lines to skip between logically adjacent lines FormatDesc **For 2D Non-Array Surfaces accessed via the Sampling Engine or Data Port:** Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures. **For Other Surfaces:** Vertical Line Stride must be zero. **Programming Notes** This bit must not be set if the surface format is a compressed type (BCn*). If this bit is set on a sampling engine surface, texture addess control modes cannot be set to any mode other than TEXCOORDMODE_CLAMP and the mip mode filter must be set to MIPFILTER_NONE.
	11	**Vertical Line Stride Offset** Project: All Format: U1 in lines of initial offset (when Vertical Line Stride == 1) FormatDesc **For 2D Non-Array Surfaces accessed via the Sampling Engine or Data Port:** Specifies the offset of the initial line from the beginning of the buffer. Ignored when Vertical **Line Stride** is 0. **For Other Surfaces:** Vertical Line Stride Offset must be zero.

SURFACE_STATE		
	10	**MIP Map Layout Mode** Project: All Format: U1 enumerated type FormatDesc **For 1D and 2D Surfaces and** This field specifies which MIP map layout mode is used, whether the map for LOD 1 is stored to the right of the LOD 0 map, or stored below it. See Memory Data Formats for details on the specifics of each layout mode. **For Other Surfaces:** This field is reserved : MBZ
	9	Reserved : MBZ

Field 10 value table:

Value	Name	Description	Project
0h	MIPLAYOUT_BELOW		All
1h	MIPLAYOUT_RIGHT		All

Programming Notes
MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces

Errata	Description	Project
#	MIPLAYOUT_RIGHT is not supported with "ld" sampler message	All
#	MIPLAYOUT_RIGHT is not supported with sample_c/sample_l_c/sample_b_c sampler messages.	[DevCL]

SURFACE_STATE		
	8	**Render Cache Read Write Mode** Project: All Format: U1 enumerated type FormatDesc **For Surfaces accessed via the Data Port to Render Cache:** This field specifies the way Render Cache treats a write request. If unset, Render Cache allocates a write-only cache line for a write miss. If set, Render Cache allocates a read-write cache line for a write miss. **For Surfaces accessed via the Sampling Engine or Data Port to Texture Cache or Data Cache:** This field is reserved : MBZ

Value	Name	Description	Project
0h		Allocating write-only cache for a write miss	All
1h		Allocating read-write cache for a write miss	All

Programming Notes
This field is provided for performance optimization for Render Cache read/write accesses (from Gen4 EU's point of view).

Errata	Description	Project
#	This field must be set to 0h.	[DevBW-A,B]

SURFACE_STATE

	7:6	**Media Boundary Pixel Mode** Project: All Format: U2 enumerated type FormatDesc **For 2D Non-Array Surfaces accessed via the Data Port Media Block Read Message:** This field enables control of which rows are returned on vertical out-of-bounds reads using the Data Port Media Block Read Message. In the description below, frame mode refers to **Vertical Line Stride** = 0, field mode is **Vertical Line Stride** = 1 in which only the even or odd rows are addressable. The frame refers to the entire surface, while the field refers only to the even or odd rows within the surface. Refer to Section 5.6.1 for more details. **For Other Surfaces:** Reserved : MBZ

Value	Name	Description	Project
0h	NORMAL_MODE	the row returned on an out-of-bound access is the closest row in the frame or field. Rows from the opposite field are never returned.	All
1h	Reserved		
2h	Reserved		
3h	Reserved		

SURFACE_STATE

	5:0	**Cube Face Enables** Project: All Format: U6 bit mask of enables FormatDesc **For SURFTYPE_CUBE Surfaces accessed via the Sampling Engine:** Bits 5:0 of this field enable the individual faces of a cube map. Enabling a face indicates that the face is present in the cube map, while disabling it indicates that that face is represented by the texture map's border color. Refer to Memory Data Formats for the correlation between faces and the cube map memory layout. Note that storage for disabled faces must be provided. **For other surfaces:** This field is reserved : MBZ

Value	**Name**	**Description**	**Project**
100000b		-X face	All
010000b		+X face	All
001000b		-Y face	All
000100b		+Y face	All
000010b		-Z face	All
000001b		+Z face	All

Programming Notes
When TEXCOORDMODE_CLAMP is used when accessing a cube map, this field must be programmed to 111111b (all faces enabled).
This field is ignored unless the **Surface Type** is SURFTYPE_CUBE.

SURFACE_STATE		
1	31:0	**Surface Base Address** Project: All Format: GraphicsAddress[31:0] FormatDesc Specifies the byte-aligned base address of the surface. **Programming Notes** For SURFTYPE_BUFFER render targets, this field specifies the base address of first element of the surface. The surface is interpreted as a simple array of that single element type. The address must be naturally-aligned to the element size (e.g., a buffer containing R32G32B32A32_FLOAT elements must be 16-byte aligned). For SURFTYPE_BUFFER non-rendertarget surfaces, this field specifies the base address of the first element of the surface, computed in software by adding the surface base address to the byte offset of the element in the buffer. Mipmapped, cube and 3D sampling engine surfaces are stored in a “monolithic” (fixed) format, and only require a single address for the base texture. Linear *render target* surface base addresses must be element-size aligned, for non-YUV surface formats, or a multiple of 2 element-sizes for YUV surface formats. Other linear surfaces have no alignment requirements (byte alignment is sufficient.) Linear depth buffer surface base addresses must be 64-byte aligned. Note that while render targets (color) can be SURFTYPE_BUFFER, depth buffers cannot. Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from **Surface Base Address** are tiled, **Surface Base Address** itself is not transformed using the tiling algorithm. Certain message types used to access surfaces have more stringent alignment requirements. Please refer to the specific message documentation for additional restrictions.

SURFACE_STATE		
2	31:19	**Height** Project: All Format: U13 FormatDesc Range SURFTYPE_1D: must be zero SURFTYPE_2D: height of surface – 1 (y/v dimension) [0,8191] SURFTYPE_3D: height of surface – 1 (y/v dimension) [0,2047] SURFTYPE_CUBE: height of surface – 1 (y/v dimension) [0,8191] SURFTYPE_BUFFER: contains bits [19:7] of the number of entries in the buffer – 1 [0,8191] This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. For buffers, this field specifies a portion of the buffer size.

Programming Notes
For buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27}. After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the **Height**, **Width**, and **Depth** fields as indicated, right-justified in each field. Unused upper bits must be set to zero.
If **Vertical Line Stride** is 1, this field indicates the height of the field, not the height of the frame
The **Height** of a render target must be the same as the **Height** of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped).

Errata	Description	Project
#	The number of entries in a SURFTYPE_BUFFER is restricted to 2^27 – 1	[DevBW-A,B]

SURFACE_STATE		
	18:6	**Width** Project: All Format: U13 FormatDesc Range SURFTYPE_1D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_2D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_3D: width of surface – 1 (x/u dimension) [0,2047] SURFTYPE_CUBE: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_BUFFER: contains bits [6:0] of the number of entries in the buffer – 1 [0,127] This field specifies the width of the surface. If the surface is MIP-mapped, this field specifies the width of the base MIP level. The width is specified in units of pixels or texels. For buffers, this field specifies a portion of the buffer size. For surfaces accessed with the Media Block Read/Write message, this field is in units of DWords. **Programming Notes** The Width specified by this field must be less than or equal to the surface pitch (specified in bytes via the **Surface Pitch** field). For cube maps, Width must be set equal to the Height. For MONO8 textures, Width must be a multiple of 32 texels. The **Width** of a render target must be the same as the **Width** of the other render target(s) and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped). The **Width** of a render target with YUV surface format must be a multiple of 2.

SURFACE_STATE		
	5:2	**MIP Count / LOD** Project: All Format: Sampling Engine Surfaces: U4 in (LOD units – 1) FormatDesc Render Target Surfaces: U4 in LOD units Range Sampling Engine Surfaces: [0,13] representing [1,14] MIP levels Render Target Surfaces: [0,13] representing LOD Other Surfaces: [0] **For Sampling Engine Surfaces:** This field indicates the number of MIP levels allowed to be accessed starting at **Surface Min LOD**, which must be less than or equal to the number of MIP levels actually stored in memory for this surface. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here. **For Render Target Surfaces:** This field defines the MIP level that is currently being rendered into. This is the absolute MIP level on the surface and is not relative to the **Surface Min LOD** field, which is ignored for render target surfaces. **For Other Surfaces:** This field is reserved : MBZ
	1:0	Reserved : MBZ

Field 5:2 (continued):

Value	Name	Description	Project
0h	Disable	Desc	All
1h	Enable	Desc	All

Programming Notes
The **LOD** of a render target must be the same as the **LOD** of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).
For render targets with YUV surface formats, the **LOD** must be zero.

Errata	Description	Project
#	Desc	All

SURFACE_STATE		
3	31:21	**Depth** Project: All Format: U11 FormatDesc Range: SURFTYPE_1D: number of array elements – 1 [0,511] SURFTYPE_2D: number of array elements – 1 [0,511] SURFTYPE_3D: depth of surface – 1 (z/r dimension) [0,2047] SURFTYPE_CUBE: number of array elements – 1 [see programming notes for range] SURFTYPE_BUFFER: contains bits [26:20] of the number of entries in the buffer – 1 [0,127] This field specifies the total number of levels for a volume texture or the number of array elements allowed to be accessed starting at the **Minimum Array Element** for arrayed surfaces. If the volume texture is MIP-mapped, this field specifies the depth of the base MIP level. For buffers, this field specifies a portion of the buffer size. **Programming Notes** The **Depth** of a render target must be the same as the **Depth** of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER). For SURFTYPE_CUBE: for all cube surfaces, this field must be zero as cube arrays are not supported.
	20	**Reserved** Project: All Format: MBZ
	19:3	**Surface Pitch** Project: All Format: U17 pitch in (#Bytes – 1) FormatDesc Range: For surfaces of type SURFTYPE_BUFFER: [0,2047] -> [1B, 2048B] For other linear surfaces: [0, 131071] -> [1B, 128KB] For X-tiled surface: [511, 131071] –> [512B, 128KB] = [1tile, 256 tiles] For Y-tiled surfaces: [127, 131071]->[128B,128KB] = [1 tile, 1024 tiles] This field specifies the surface pitch in (#Bytes - 1). For surfaces of type SURFTYPE_BUFFER, this field indicates the size of the structure. **Programming Notes** For linear *render target* surfaces, the pitch must be a multiple of the element size for non-YUV surface formats. Pitch must be a multiple of 2 * element size for YUV surface formats. For other linear surfaces, the pitch can be any multiple of bytes. For tiled surfaces, the pitch must be a multiple of the tile width.
	2	**Reserved** Project: All Format: MBZ

SURFACE_STATE

	1	**Tiled Surface**

Tiled Surface

Project: All

Format: U1 enumerated type FormatDesc

This field specifies whether the surface is tiled.

Value	Name	Description	Project
0h	FALSE	Linear surface	All
1h	TRUE	Tiled surface	All

Programming Notes
Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory.
The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.
If **Surface Type** is SURFTYPE_BUFFER, this field must be FALSE (buffers are supported only in linear memory)
If the target cache via the Data Port is the Data Cache, this field must be disabled (zero). The data cache only supports access to linear memory.
If **Surface Type** is SURFTYPE_NULL, this field must be TRUE

0 **Tile Walk**

Project: All

Format: U1 enumerated type FormatDesc

This field specifies the type of memory tiling (XMajor or YMajor) employed to tile this surface. See *Memory Interface Functions* for details on memory tiling and restrictions.

Value	Name	Description	Project
0h	TILEWALK_XMAJOR	X major tiling	All
1h	TILEWALK_YMAJOR	Y major tiling	All

Programming Notes
Refer to *Memory Data Formats* for restrictions on *TileWalk* direction for the various buffer types. (Of particular interest is the fact that YMAJOR tiling is **not** supported for display/overlay buffers).
The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.
Use of TILEWALK_YMAJOR is UNDEFINED for render target formats that have 128 bits-per-element (BPE).
This field is ignored when the surface is linear.

SURFACE_STATE		
4	31:28	**Surface Min LOD** Project: All Format: U4 in LOD units FormatDesc Range [0,13] **For Sampling Engine Surfaces:** This field indicates the most detailed LOD that can be accessed as part of this surface. This field is added to the delivered LOD (sample_l, ld, or resinfo message types) before it is used to address the surface. **For Other Surfaces:** This field is ignored. **Programming Notes** This field must be zero if the **Surface Format** is MONO8 **[DevBW-A,B]:** this field must be zero
	27:17	**Minimum Array Element** Project: All Format: U11 FormatDesc Range 1D/2D/cube surfaces: [0,511] 3D surfaces: [0,2047] **For Sampling Engine and Render Target 1D and 2D Surfaces:** This field indicates the minimum array element that can be accessed as part of this surface. This field is added to the delivered array index before it is used to address the surface. **For Render Target 3D Surfaces:** This field indicates the minimum 'R' coordinate on the LOD currently being rendered to. This field is added to the delivered array index before it is used to address the surface. **For Other Surfaces:** This field must be set to zero.

Errata	Description	Project
#	This field must be zero.	[DevBW-A,B]
#	For sample_c/sample_b_c/sample_l_c instructions this field is ignored. If it is **tiled surface** and not at a 4k boundary it must be copied to a 4k aligned surface. Then for any case it must be pointed to by the **Surface Base Address.**	[DevBW-A,B,C,D], [DevCL-A,B]

SURFACE_STATE		
	16:8	**Render Target View Extent** Project: All Format: U9 FormatDesc Range [0,511] to indicate extent of [1,512] **For Render Target 3D Surfaces:** This field indicates the extent of the accessible 'R' coordinates minus 1 on the LOD currently being rendered to. **For Render Target 1D and 2D Surfaces:** This field must be set to the same value as the **Depth** field. **For Other Surfaces:** This field is ignored.
	7	**Reserved** Project: All Format: MBZ
	6:4	**Number of Multisamples.** Reserved : MBZ
	3	**Reserved** Project: All Format: MBZ
	2:0	Reserved : MBZ

4.7.2.1.1 Surface Formats

The following table indicates the supported surface formats and the 9-bit encoding for each. Note that some of these formats are used not only by the Sampling Engine, but also by the Data Port and the Vertex Fetch unit.

Support of each format and capability is as follows:

Y	supported on all products
Y*	Not used in PRM
Y+	Not used in PRM
Y~	Not used in PRM
Y^	Not used in PRM

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	🟩			Y	Y	Y	Y		000	R32G32B32A32_FLOAT	128**
Y				Y		Y	Y		001	R32G32B32A32_SINT	128**
Y				Y		Y	Y		002	R32G32B32A32_UINT	128**
						Y			003	R32G32B32A32_UNORM	128
						Y			004	R32G32B32A32_SNORM	128
						Y			005	R64G64_FLOAT	128
Y	🟩								006	R32G32B32X32_FLOAT	128
						Y			007	R32G32B32A32_SSCALED	128
						Y			008	R32G32B32A32_USCALED	128
Y	🟩					Y	Y		040	R32G32B32_FLOAT	96
Y						Y	Y		041	R32G32B32_SINT	96
Y						Y	Y		042	R32G32B32_UINT	96
						Y			043	R32G32B32_UNORM	96
						Y			044	R32G32B32_SNORM	96
						Y			045	R32G32B32_SSCALED	96
						Y			046	R32G32B32_USCALED	96
Y	Y			Y	🟥	Y		🟩	080	R16G16B16A16_UNORM	64
Y	Y			Y	🟩	Y			081	R16G16B16A16_SNORM	64
Y				Y		Y			082	R16G16B16A16_SINT	64
Y				Y		Y			083	R16G16B16A16_UINT	64
Y	Y			Y	Y	Y			084	R16G16B16A16_FLOAT	64
Y	🟩			Y	Y	Y	Y		085	R32G32_FLOAT	64
Y				Y		Y	Y		086	R32G32_SINT	64
Y				Y		Y	Y		087	R32G32_UINT	64
Y	🟩	Y							088	R32_FLOAT_X8X24_TYPELESS	64
Y									089	X32_TYPELESS_G8X24_UINT	64
Y	🟩								08A	L32A32_FLOAT	64
						Y			08B	R32G32_UNORM	64
						Y			08C	R32G32_SNORM	64
						Y			08D	R64_FLOAT	64
Y	Y								08E	R16G16B16X16_UNORM	64
Y	Y								08F	R16G16B16X16_FLOAT	64
Y	🟩								090	A32X32_FLOAT	64

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	🟩								091	L32X32_FLOAT	64
Y	🟩								092	I32X32_FLOAT	64
						Y			093	R16G16B16A16_SSCALED	64
						Y			094	R16G16B16A16_USCALED	64
						Y			095	R32G32_SSCALED	64
						Y			096	R32G32_USCALED	64
Y	Y		Y	Y	Y	Y			0C0	B8G8R8A8_UNORM	32
Y	Y			Y	Y				0C1	B8G8R8A8_UNORM_SRGB	32
Y	Y			Y	Y	Y		🟩	0C2	R10G10B10A2_UNORM	32
Y	Y							🟩	0C3	R10G10B10A2_UNORM_SRGB	32
Y				Y		Y			0C4	R10G10B10A2_UINT	32
Y	Y					Y			0C5	R10G10B10_SNORM_A2_UNORM	32
Y	Y			Y	Y	Y		🟩	0C7	R8G8B8A8_UNORM	32
Y	Y			Y	Y			🟩	0C8	R8G8B8A8_UNORM_SRGB	32
Y	Y			Y	🟩	Y			0C9	R8G8B8A8_SNORM	32
Y				Y		Y			0CA	R8G8B8A8_SINT	32
Y				Y		Y			0CB	R8G8B8A8_UINT	32
Y	Y			Y	🟥	Y			0CC	R16G16_UNORM	32
Y	Y			Y	🟩	Y			0CD	R16G16_SNORM	32
Y				Y		Y			0CE	R16G16_SINT	32
Y				Y		Y			0CF	R16G16_UINT	32
Y	Y			Y	Y	Y			0D0	R16G16_FLOAT	32
Y	Y			Y	Y			🟩	0D1	B10G10R10A2_UNORM	32
Y	Y			Y	Y			🟩	0D2	B10G10R10A2_UNORM_SRGB	32
Y	Y			Y	Y	Y			0D3	R11G11B10_FLOAT	32
Y				Y		Y	Y		0D6	R32_SINT	32
Y				Y		Y	Y		0D7	R32_UINT	32
Y	🟩	Y		Y	Y	Y	Y		0D8	R32_FLOAT	32
Y	🟩	Y							0D9	R24_UNORM_X8_TYPELESS	32
Y									0DA	X24_TYPELESS_G8_UINT	32
Y	Y								0DF	L16A16_UNORM	32
Y	🟩	Y							0E0	I24X8_UNORM	32
Y	🟩	Y							0E1	L24X8_UNORM	32

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	🟩	Y							0E2	A24X8_UNORM	32
Y	🟩	Y							0E3	I32_FLOAT	32
Y	🟩	Y							0E4	L32_FLOAT	32
Y	🟩	Y							0E5	A32_FLOAT	32
Y	Y		Y						0E9	B8G8R8X8_UNORM	32
Y	Y								0EA	B8G8R8X8_UNORM_SRGB	32
Y	Y								0EB	R8G8B8X8_UNORM	32
Y	Y								0EC	R8G8B8X8_UNORM_SRGB	32
Y	Y								0ED	R9G9B9E5_SHAREDEXP	32
Y	Y								0EE	B10G10R10X2_UNORM	32
Y	Y								0F0	L16A16_FLOAT	32
						Y			0F1	R32_UNORM	32
						Y			0F2	R32_SNORM	32
						Y			0F3	R10G10B10X2_USCALED	32
						Y			0F4	R8G8B8A8_SSCALED	32
						Y			0F5	R8G8B8A8_USCALED	32
						Y			0F6	R16G16_SSCALED	32
						Y			0F7	R16G16_USCALED	32
						Y			0F8	R32_SSCALED	32
						Y			0F9	R32_USCALED	32
Y	Y		Y	Y	Y				100	B5G6R5_UNORM	16
Y	Y			Y	Y				101	B5G6R5_UNORM_SRGB	16
Y	Y		Y	Y	Y				102	B5G5R5A1_UNORM	16
Y	Y			Y	Y				103	B5G5R5A1_UNORM_SRGB	16
Y	Y		Y	Y	Y				104	B4G4R4A4_UNORM	16
Y	Y			Y	Y				105	B4G4R4A4_UNORM_SRGB	16
Y	Y			Y	Y	Y			106	R8G8_UNORM	16
Y	Y		Y	Y	🟩	Y			107	R8G8_SNORM	16
Y				Y		Y			108	R8G8_SINT	16
Y				Y		Y			109	R8G8_UINT	16
Y	Y	Y		Y	🟥	Y			10A	R16_UNORM	16
Y	Y			Y	🟩	Y			10B	R16_SNORM	16
Y				Y		Y			10C	R16_SINT	16

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y				Y		Y			10D	R16_UINT	16
Y	Y			Y	Y	Y			10E	R16_FLOAT	16
									10F	A8P8_UNORM [palette0]	16
									110	A8P8_UNORM [palette1]	16
Y	Y	Y							111	I16_UNORM	16
Y	Y	Y							112	L16_UNORM	16
Y	Y	Y							113	A16_UNORM	16
Y	Y		Y						114	L8A8_UNORM	16
Y	Y	Y							115	I16_FLOAT	16
Y	Y	Y							116	L16_FLOAT	16
Y	Y	Y							117	A16_FLOAT	16
									118	L8A8_UNORM_SRGB	16
Y	Y		Y						119	R5G5_SNORM_B6_UNORM	16
				Y	Y				11A	B5G5R5X1_UNORM	16
				Y	Y				11B	B5G5R5X1_UNORM_SRGB	16
						Y			11C	R8G8_SSCALED	16
						Y			11D	R8G8_USCALED	16
						Y			11E	R16_SSCALED	16
						Y			11F	R16_USCALED	16
									122	P8A8_UNORM [palette0]	16
									123	P8A8_UNORM [palette1]	16
Y	Y			Y	Y	Y			140	R8_UNORM	8
Y	Y			Y		Y			141	R8_SNORM	8
Y				Y		Y			142	R8_SINT	8
Y				Y		Y			143	R8_UINT	8
Y	Y		Y	Y	Y				144	A8_UNORM	8
Y	Y								145	I8_UNORM	8
Y	Y		Y						146	L8_UNORM	8
Y	Y								147	P4A4_UNORM [palette0]	8
Y	Y								148	A4P4_UNORM [palette0]	8
						Y			149	R8_SSCALED	8
						Y			14A	R8_USCALED	8
									14B	P8_UNORM [palette0]	8

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
🟨	🟨								14C	L8_UNORM_SRGB	8
🟥	🟥								14D	P8_UNORM [palette1]	8
🟥	🟥								14E	P4A4_UNORM [palette1]	8
🟥	🟥								14F	A4P4_UNORM [palette1]	8
🟨	🟨								180	BC1_RGB_SRGB	0
Y	Y								181	R1_UNORM/R1_UINT	1
Y	Y		Y	Y				🟩	182	YCRCB_NORMAL	0
Y	Y		Y	Y				🟩	183	YCRCB_SWAPUVY	0
🟨	🟨								184	P2_UNORM [palette0]	2
🟥	🟥								185	P2_UNORM [palette1]	2
Y	Y		Y						186	BC1_UNORM	0
Y	Y		Y						187	BC2_UNORM	0
Y	Y		Y						188	BC3_UNORM	0
Y	Y								18B	BC1_UNORM_SRGB	0
Y	Y								18C	BC2_UNORM_SRGB	0
Y	Y								18D	BC3_UNORM_SRGB	0
Y									18E	MONO8	1
Y	Y			Y				🟩	18F	YCRCB_SWAPUV	0
Y	Y			Y				🟩	190	YCRCB_SWAPY	0
Y	Y								191	BC1_RGB	0
Y	Y								192	FXT1	0
						Y			193	R8G8B8_UNORM	24
						Y			194	R8G8B8_SNORM	24
						Y			195	R8G8B8_SSCALED	24
						Y			196	R8G8B8_USCALED	24
						Y			197	R64G64B64A64_FLOAT	256
						Y			198	R64G64B64_FLOAT	192
🟧	🟧								19B	R16G16B16_FLOAT	48
						Y			19C	R16G16B16_UNORM	48
						Y			19D	R16G16B16_SNORM	48
						Y			19E	R16G16B16_SSCALED	48
						Y			19F	R16G16B16_USCALED	48

** Note: 128 BPE Formats cannot be Tiled Y when used as render targets

4.7.2.1.2 Sampler Output Channel Mapping

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

Surface Format Name		R	G	B	A
R32G32B32X32_FLOAT		R	G	B	1.0
R32G32B32_FLOAT		R	G	B	1.0
R32G32B32_SINT		R	G	B	1.0
R32G32B32_UINT		R	G	B	1.0
R32G32_FLOAT		R	G	1.0	1.0
		R	G	0.0	1.0
R32G32_SINT		R	G	0.0	1.0
R32G32_UINT		R	G	0.0	1.0
R32_FLOAT_X8X24_TYPELESS		R	0.0	0.0	1.0
X32_TYPELESS_G8X24_UINT		0.0	G	0.0	1.0
L32A32_FLOAT		L	L	L	A
R16G16B16X16_UNORM		R	G	B	1.0
R16G16B16X16_FLOAT		R	G	B	1.0
A32X32_FLOAT		0.0	0.0	0.0	A
L32X32_FLOAT		L	L	L	1.0
I32X32_FLOAT		I	I	I	I
R16G16_UNORM		R	G	1.0	1.0
		R	G	0.0	1.0
R16G16_SNORM		R	G	1.0	1.0
		R	G	0.0	1.0
R16G16_SINT		R	G	0.0	1.0
R16G16_UINT		R	G	0.0	1.0
R16G16_FLOAT		R	G	1.0	1.0
		R	G	0.0	1.0
R11G11B10_FLOAT		R	G	B	1.0
R32_SINT		R	0.0	0.0	1.0
R32_UINT		R	0.0	0.0	1.0

Surface Format Name		R	G	B	A
R32_FLOAT		R	1.0	1.0	1.0
		R	0.0	0.0	1.0
R24_UNORM_X8_TYPELESS		R	0.0	0.0	1.0
X24_TYPELESS_G8_UINT		0.0	G	0.0	1.0
L16A16_UNORM		L	L	L	A
I24X8_UNORM		I	I	I	I
L24X8_UNORM		L	L	L	1.0
A24X8_UNORM		0.0	0.0	0.0	A
I32_FLOAT		I	I	I	I
L32_FLOAT		L	L	L	1.0
A32_FLOAT		0.0	0.0	0.0	A
B8G8R8X8_UNORM		R	G	B	1.0
B8G8R8X8_UNORM_SRGB		R	G	B	1.0
R8G8B8X8_UNORM		R	G	B	1.0
R8G8B8X8_UNORM_SRGB		R	G	B	1.0
R9G9B9E5_SHAREDEXP		R	G	B	1.0
B10G10R10X2_UNORM		R	G	B	1.0
L16A16_FLOAT		L	L	L	A
B5G6R5_UNORM		R	G	B	1.0
B5G6R5_UNORM_SRGB		R	G	B	1.0
R8G8_UNORM		R	G	1.0	1.0
		R	G	0.0	1.0
R8G8_SNORM		R	G	1.0	1.0
		R	G	0.0	1.0
R8G8_SINT		R	G	0.0	1.0
R8G8_UINT		R	G	0.0	1.0
R16_UNORM		R	0.0	0.0	1.0
R16_SNORM		R	0.0	0.0	1.0
R16_SINT		R	0.0	0.0	1.0
R16_UINT		R	0.0	0.0	1.0
R16_FLOAT		R	1.0	1.0	1.0
		R	0.0	0.0	1.0
I16_UNORM		I	I	I	I
L16_UNORM		L	L	L	1.0

Surface Format Name		R	G	B	A
A16_UNORM		0.0	0.0	0.0	A
L8A8_UNORM		L	L	L	A
I16_FLOAT		I	I	I	I
L16_FLOAT		L	L	L	1.0
A16_FLOAT		0.0	0.0	0.0	A
R5G5_SNORM_B6_UNORM		R	G	B	1.0
R8_UNORM		R	0.0	0.0	1.0
R8_SNORM		R	0.0	0.0	1.0
R8_SINT		R	0.0	0.0	1.0
R8_UINT		R	0.0	0.0	1.0
A8_UNORM		0.0	0.0	0.0	1.0
I8_UNORM		I	I	I	I
L8_UNORM		L	L	L	1.0
L8_UNORM_SRGB		L	L	L	1.0
R1_UNORM/R1_UINT		R	0.0	0.0	1.0
YCRCB_NORMAL		Cr	Y	Cb	1.0
YCRCB_SWAPUVY		Cr	Y	Cb	1.0
YCRCB_SWAPUV		Cr	Y	Cb	1.0
YCRCB_SWAPY		Cr	Y	Cb	1.0
BC1_RGB		R	G	B	1.0
BC1_RGB_SRGB		R	G	B	1.0

4.7.3 SAMPLER_STATE

SAMPLER_STATE has three different formats, depending on the message type used. All messages use the format described under "For most messages".

4.7.3.1 For Most Messages

SAMPLER_STATE	
Project:	All
This is the normal sampler state used by all messages that use SAMPLER_STATE except *sample_8x8* and *deinterlace*. The sampler state is stored as an array of up to 16 elements, each of which contains the dwords described here. The start of each element is spaced 4 dwords apart. The first element of the sampler state array is aligned to a 32-byte boundary.	

SAMPLER_STATE		
DWord	**Bit**	**Description**
0	31	**Sampler Disable** Project: All Format: Disable FormatDesc This field allows the sampler to be disabled. If disabled, all output channels will return 0.
	30:29	**Reserved** Project: All Format: MBZ
	28	**LOD PreClamp Enable** Project: All Format: U1 enumerated type FormatDesc When enabled, the computed LOD is clamped to [max,min] mip level *before* the mag-vs-min determination is performed. This is how the standard API currently performs min/mag determination, and therefore it is expected that a standard API driver would need to set this bit.
	27 26:22	**Reserved** Project: All Format: MBZ **Base Mip Level** Project: All Format: U4.1 FormatDesc Range [0.0,13.0] Specifies which mip level is considered the "base" level when determining mag-vs-min filter and selecting the "base" mip level.

SAMPLER_STATE		
	21:20	**Mip Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines if and how mip map levels are chosen and/or combined when texture filtering.

Value	Name	Description	Project
0h	MIPFILTER_NONE	Disable mip mapping – force use of the mipmap level corresponding to **Min LOD**.	All
1h	MIPFILTER_NEAREST	Nearest, Select the nearest mip map	All
2h	Reserved		All
3h	MIPFILTER_LINEAR	Linearly interpolate between nearest mip maps (combined with linear min/mag filters this is analogous to "Trilinear" filtering).	All

Programming Notes
MIPFILTER_LINEAR is not supported for surface formats that do not support "Sampling Engine Filtering" as indicated in the Surface Formats table unless using the sample_c message type.

SAMPLER_STATE

	19:17	**Mag Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines how texels are sampled/filtered when a texture is being "magnified" (enlarged). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension.

Value	**Name**	**Description**	**Project**
0h	MAPFILTER_NEAREST	Sample the nearest texel	All
1h	MAPFILTER_LINEAR	Bilinearly filter the 4 nearest texels	All
2h	MAPFILTER_ANISOTROPIC	Perform an "anisotropic" filter on the chosen mip level	All
3h-5h	Reserved		All
6h	MAPFILTER_MONO	Perform a monochrome convolution filter	All
7h	Reserved		All

Programming Notes
Only MAPFILTER_NEAREST and MAPFILTER_LINEAR are supported for surfaces of type SURFTYPE_3D.
Only MAPFILTER_NEAREST is supported for surface formats that do not support "Sampling Engine Filtering" as indicated in the Surface Formats table unless using the sample_c message type.
MAPFILTER_MONO: Only CLAMP_BORDER texture addressing mode is supported. . Both **Mag Mode Filter** and **Min Mode Filter** must be programmed to MAPFILTER_MONO. **Mip Mode Filter** must be MIPFILTER_NONE. Only valid on surfaces with **Surface Format** MONO8 and with **Surface Type** SURFTYPE_2D.
MAPFILTER_ANISOTROPIC may cause artifacts at cube edges if enabled for cube maps with the TEXCOORDMODE_CUBE addressing mode.
MAPFILTER_ANISOTROPIC will be overridden to MAPFILTER_LINEAR when using a sample_l or sample_l_c message type or when **Force LOD to Zero** is set in the message header. **[DevBW, DevCL] Errata: Force LOD to Zero** will not cause MAPFILTER_ANISOTROPIC to get forced to MAPFILTER_LINEAR and instead it will have to be worked around using sample_l or sample_l_c.

	16:14	**Min Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines how texels are sampled/filtered when a texture is being "minified" (shrunk). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension. See **Mag Mode Filter**

SAMPLER_STATE

	13:3	**Texture LOD Bias** Project: All Format: S4.6 2's complement FormatDesc Range [-16.0, 16.0) This field specifies the signed bias value added to the calculated texture map LOD prior to min-vs-mag determination and mip-level clamping. Assuming mipmapping is enabled, a positive LOD bias will result in a somewhat blurrier image (using less-detailed mip levels) and possibly higher performance, while a negative bias will result in a somewhat crisper image (using more-detailed mip levels) and may lower performance. **Programming Notes** There is no requirement or need to offset the LOD Bias in order to produce a correct LOD for texture filtering (as was required for correct bilinear and anisotropic filtering in some legacy devices).
	2:0	**Shadow Function** Project: All Format: U3 enumerated type FormatDesc This field is used for shadow mapping support via the sample_c message type, and specifies the specific comparison operation to be used. The comparison is between the texture sample red channel (except for alpha-only formats which use the alpha channel), and the "ref" value provided in the input message.

Value	Name	Description	Project
0h	PREFILTEROP_ALWAYS		All
1h	PREFILTEROP_NEVER		All
2h	PREFILTEROP_LESS		All
3h	PREFILTEROP_EQUAL		All
4h	PREFILTEROP_LEQUAL		All
5h	PREFILTEROP_GREATER		All
6h	PREFILTEROP_NOTEQUAL		All
7h	PREFILTEROP_GEQUAL		All

SAMPLER_STATE		
1	31:22	**Min LOD** Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0], where the upper limit is also bounded by the **Max LOD**. This field specifies the minimum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this maximum (resolution) mip clamping is applied. The integer bits of this field are used to control the "maximum" (highest resolution) mipmap level that may be accessed (where LOD 0 is the highest resolution map). The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. **Programming Notes** If **Min LOD** is greater than **Max LOD**, **Min LOD** takes precedence, i.e. the resulting LOD will always be **Min LOD**. This field must be zero if the **Min** or **Mag Mode Filter** is set to MAPFILTER_MONO **Errata** / **Description** / **Project** # / If the **Mip Mode Filter** is set to MIPFILTER_NEAREST and the fractional portion of **MIn LOD** is < 0.5 but > 0.0, the LOD chosen is one too large. Zeroing the fractional portion of **Min LOD** in these cases gives the correct behavior as a software workaround. / All
	21:12	**Max LOD** Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0] This field specifies the maximum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this minimum (resolution) mip clamping is applied. The integer bits of this field are used to control the "minimum" (lowest resolution) mipmap level that may be accessed. The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here. **Programming Notes** If **Min LOD** is greater than **Max LOD**, **Min LOD** takes precedence, i.e. the resulting LOD will always be **Min LOD**.
	11:10	**Reserved** Project: All Format: MBZ

SAMPLER_STATE		
	9	**Cube Surface Control Mode** Project: All Format: U1 enumerated type FormatDesc When sampling from a SURFTYPE_CUBE surface, this field controls whether the **TC* Address Control Mode** fields are interpreted as programmed or overridden to TEXCOORDMODE_CUBE.

Value	Name	Description	Project
0h	CUBECTRLMODE_PROGRAMMED		All
1h	CUBECTRLMODE_OVERRIDE		All

Errata	Description	Project
#	this field must be set to CUBECTRLMODE_PROGRAMMED	[DevBW-A,B], [DevCL-A]

SAMPLER_STATE

	8:6	**TCX Address Control Mode** Project: All Format: U3 enumerated type FormatDesc Controls how the 1st (TCX, aka U) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates "outside" the texture are handled (wrap/clamp/mirror). The setting of this field is subject to being overridden by the **Cube Surface Control Mode** field when sampling from a SURFTYPE_CUBE surface.

Value	Name	Description	Project
0h	TEXCOORDMODE_WRAP	Map is repeated in the U direction	All
1h	TEXCOORDMODE_MIRROR	Map is mirrored in the U direction	All
2h	TEXCOORDMODE_CLAMP	Map is clamped to the edges of the accessed map	All
3h	TEXCOORDMODE_CUBE	For cube-mapping, filtering in edges access adjacent map faces	All
4h	TEXCOORDMODE_CLAMP_BORDER	Map is infinitely extended with the border color	All
5h	TEXCOORDMODE_MIRROR_ONCE	Map is mirrored once about origin, then clamped	All
6h-7h	Reserved		All

Programming Notes
When using cube map texture coordinates, only TEXCOORDMODE_CLAMP and TEXCOORDMODE_CUBE settings are valid, and each TC component must have the same Address Control mode.
When TEXCOORDMODE_CLAMP is used when accessing a cube map, the map's **Cube Face Enable** field must be programmed to 111111b (all faces enabled).
MAPFILTER_MONO: Texture addressing modes must all be set to TEXCOORDMODE_CLAMP_BORDER. The **Border Color** is ignored in this mode, a constant value of 0 is used for border color. Software must pad the border texels within the map itself with 0.

	5:3	**TCY Address Control Mode** Project: All Format: U3 enumerated type FormatDesc Controls how the 2nd (TCY, aka V) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates "outside" the texture are handled (wrap/clamp/mirror) or whether the "wrap shortest" mapping should be applied. See **Address TCX Control Mode** above for details

SAMPLER_STATE		
	2:0	**TCZ Address Control Mode** Project: All Format: U3 enumerated type FormatDesc Controls how the 3rd (TCZ) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates "outside" the texture are handled (wrap/clamp/mirror) or whether the "wrap shortest" mapping should be applied. See **Address TCX Control Mode** above for details
2	31:5	**Border Color Pointer** Project: All Format: GeneralStateOffset[31:5] FormatDesc This field specifies the pointer to SAMPLER_BORDER_COLOR_STATE, which contains the "border" color to be used when accessing texels not contained within the texture map. This pointer is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
3	31:29	**Monochrome Filter Height** Project: All Format: U3 FormatDesc Range [1,7] This field specifies the height of the monochrome filter. It is ignored if the monochrome filter is not enabled.
	28:26	**Monochrome Filter Width** Project: All Format: U3 FormatDesc Range [1,7] This field specifies the width of the monochrome filter. It is ignored if the monochrome filter is not enabled.
	25	**ChromaKey Enable** Project: All Format: Enable FormatDesc This field enables the chroma key function. **Programming Notes** Supported only on a specific subset of surface formats. See section 4.7.2.1.1 "Surface Formats" for supported formats. This field must be disabled if min or mag filter is MAPFILTER_MONO or MAPFILTER_ANISOTROPIC. This field must be disabled if used with a surface of type SURFTYPE_3D.

SAMPLER_STATE		
	24:23	**ChromaKey Index** Project: All Format: U2 FormatDesc Range [0,3] This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a "don't care" unless **ChromaKey Enable** is ENABLED.
	22	**ChromaKey Mode** Project: All Format: U1 enumerated type FormatDesc This field specifies the behavior of the device in the event of a ChromaKey match. This field is ignored if ChromaKey is disabled. KEYFILTER_KILL_ON_ANY_MATCH: In this mode, if any contributing texel matches the chroma key, the corresponding pixel mask bit for that pixel is cleared. The result of this operation is observable only if the **Killed Pixel Mask Return** flag is set on the input message. KEYFILTER_REPLACE_BLACK: In this mode, each texel that matches the chroma key is replaced with (0,0,0,0) (black with alpha=0) prior to filtering. For YCrCb surface formats, the black value is A=0, R(Cr)=0x80, G(Y)=0x10, B(Cb)=0x80. This will tend to darken/fade edges of keyed regions. Note that the pixel pipeline must be programmed to use the resulting filtered texel value to gain the intended effect, e.g., handle the case of a totally keyed-out region (filtered texel alpha==0) through use of alpha test, etc.

Value	Name	Description	Project
0h	KEYFILTER_KILL_ON_ANY_MATCH		All
1h	KEYFILTER_REPLACE_BLACK		All

SAMPLER_STATE

	Bits	Description
	21:19	**Maximum Anisotropy** Project: All Format: U3 enumerated type FormatDesc This field clamps the maximum value of the anisotropy ratio used by the MAPFILTER_ANISOTROPIC filter (Min or Mag Mode Filter).

Value	Name	Description	Project
0h	ANISORATIO_2	At most a 2:1 aspect ratio filter is used	All
1h	ANISORATIO_4	At most a 4:1 aspect ratio filter is used	All
2h	ANISORATIO_6	At most a 6:1 aspect ratio filter is used	All
3h	ANISORATIO_8	At most a 8:1 aspect ratio filter is used	All
4h	ANISORATIO_10	At most a 10:1 aspect ratio filter is used	All
5h	ANISORATIO_12	At most a 12:1 aspect ratio filter is used	All
6h	ANISORATIO_14	At most a 14:1 aspect ratio filter is used	All
7h	ANISORATIO_16	At most a 16:1 aspect ratio filter is used	All

	Bits	Description
	18:13	**Address Rounding Enable** Project: All Format: 6-bit mask of enables FormatDesc Controls whether the U/V/R texture address is rounded or truncated before being used to select texels to sample. Each bit provides independent control of rounding on one texture address dimension (U/V/R) in either mag or min filter mode.

Value	Name	Description	Project
100000b		U address mag filter	All
010000b		U address min filter	All
001000b		V address mag filter	All
000100b		V address min filter	All
000010b		R address mag filter	All
000001b		R address min filter	All

	Bits	Description
	12:0	**Reserved** Project: All Format: MBZ

4.7.4 SAMPLER_BORDER_COLOR_STATE

This structure is pointed to by a field in SAMPLER_STATE.

The interpretation of the border color is as follows:

- The format of the border color is R32G32B32A32_FLOAT, regardless of the surface format chosen. For surface formats with one or more channels missing, the value from the border color is not used for the missing channels, resulting in these channels resulting in the overall default value (0 for colors and 1 for alpha) regardless of whether border color is chosen. The surface formats with "L" and "I" have special behavior with respect to the border color. The border color value used for the replicated channels (RGB for "L" formats and RGBA for "I" formats) comes from the *red* channel of border color. In these cases, the green and blue channels, and also alpha for "I", of the border color are ignored.

Programming Notes:

- The conditions under which this color is used depend on the **Surface Type** – 1D/2D/3D surfaces use the border color when the coordinates extend beyond the surface extent; cube surfaces use the border color for "empty" (disabled) faces.
- The border color itself is accessed through the texture cache hierarchy rather than the state cache hierarchy. Thus, if the border color is changed in memory, the texture cache must be invalidated and the state cache does not need to be invalidated.
- MAPFILTER_MONO: The border color is ignored. Border color is fixed at a value of 0 by hardware.

DWord	Bit	Description
0	31:0	**Border Color Red** Format = IEEE_FP
1	31:0	**Border Color Green** Format = IEEE_FP
2	31:0	**Border Color Blue** Format = IEEE_FP
3	31:0	**Border Color Alpha** Format = IEEE_FP

4.7.5 3DSTATE_CHROMA_KEY

3DSTATE_CHROMA_KEY			
Project:	All	**Length Bias:**	2

The 3DSTATE_CHROMA_KEY instruction is used to program texture color/chroma-key key values. A table containing four set of values is supported. The **ChromaKey Index** sampler state variable is used to select which table entry is associated with the map. Texture chromakey functions are enabled and controlled via use of the **ChromaKey Enable** texture sampler state variable.

Texture Color Key (keying on a paletted texture index) is not supported.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE Format: OpCode
	24:16	**3D Command Sub Opcode** Default Value: 04h 3DSTATE_CHROMA_KEY Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:30	**ChromaKey Table Index** Project: All Format: U2 index Range 0..3 Selects which entry in the ChromaKey table is to be loaded
	29:0	**Reserved** Project: All Format: MBZ
2	31:0	**ChromaKey Low Value** This field specifies the "low" (minimum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range. See **ChromaKey High Value** for further format, programming info.

3DSTATE_CHROMA_KEY

3	31:0	**ChromaKey High Value** This field specifies the "high" (maximum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range.

Programming Notes

ChromaKey values are specified using 8-bit channels. When using surface formats with less than 8 bits per channel, the device will expand channels by replicating the required number of MSBs into the LSBs of each channel. Software must account for this conversion when it programs Chromakey Low/High Values (e.g., by performing the same replication).

For channels that do not exist in the actual surface (e.g., Alpha channel for non-ARGB maps), software must explicitly program full range high/low values (High=FFh, Low=0h for formats using unsigned chroma key values, High=7Fh, Low=FFh for formats using sign magnitude chroma key values) in order to effectively remove the comparison of that field from the ChromaKey function.

For channels in SNORM format in the surface format, the value in the high/low value for that channel is interpreted in *sign magnitude* format. Negative zero value is not supported (use positive zero instead). For channels with mixed UNORM/SNORM formats (i.e. R5G5_SNORM_B6_UNORM), the ChromaKey is programmed as if all channels are SNORM.

YUV ChromaKey will use an interpolated chrominance value from the map for comparison to the chroma key values for those texels without chrominance due to downsampling. The chrominance value used is the average of values to the left and right of the texel in question.

It is UNDEFINED to program any component of the ChromaKey High Value to be less than the corresponding component of ChromaKey Low Value.

Format = interpreted according to associated texel format "class":

Only the surface formats listed as supported for chroma key in the surface formats table can be used with this feature. Use of any other surface format with chroma key enabled is UNDEFINED.

Surface Format	31:24	23:16	15:8	7:0
ARGB and BC formats	A	R	G	B
YCrCb formats	A	Cr	Y	Cb

4.7.6 3DSTATE_SAMPLER_PALETTE_LOAD0

3DSTATE_SAMPLER_PALETTE_LOAD0			
Project:	All	**Length Bias:**	2

The 3DSTATE_SAMPLER_PALETTE_LOAD0 instruction is used to load 24-bit values into the first texture palette. The texture palette is used whenever a texture with a paletted format (containing "Px [palette0]") is referenced by the sampler.

This instruction is used to load all or a subset of the 16 entries of the first palette. Partial loads always start from the first (index 0) entry.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE Format: OpCode
	24:16	**3D Command Sub Opcode** Default Value: 02h 3DSTATE_SAMPLER_PALETTE_LOAD0 Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1..n	31:24	Reserved : MBZ
	23:0	**Palette Color[0:N-1]** Project: All Colors loaded into the first N entries of the texture color palette. Format = Bits 23:0 = U24 interpreted as RGB_888 color as follows: [23:16] Red [15:8] Green [7:0] Blue

4.8 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.
- **[DevBW-A,B,C0, DevCL-A0] Errata:** use of any Sampling Engine message in the same workload (between pipeline flushes) with any Data Port read messages utilizing the Sampler Cache or Data Cache is not allowed.

4.8.1 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, as 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.

4.8.1.1 Message Descriptor

Bit	Description
15:14	**Message Type:** Specifies the type of message being sent, along with the message length (in the general message descriptor) Format = U2 Refer to the table in section 4.8.1.3 for encoding details.
13:12	**Data Return Format:** Specifies the format of the data returned to the requesting thread. 00 = FLOAT32 – return a signed 32-bit IEEE Float to the thread. Required for all UNORM, SNORM, and FLOAT surface formats. Can be used by resinfo messages regardless of surface format. 01 = Reserved 10 = UINT32 – return an unsigned 32-bit integer. Required for all UINT surface formats. Can be used by resinfo messages regardless of surface format. 11 = SINT32 – return a signed 32-bit 2's complement integer. Required for all SINT surface formats.

11:8	**Sampler Index:** Specifies the index into the sampler state table. Ignored for "ld" and "resinfo" type messages.
	
	Format = U4
	
	Range = [0,15]

7:0	**Binding Table Index:** Specifies the index into the binding table.
	
	Format = U8
	
	Range = [0,255]

4.8.1.2 Message Header

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

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for "ld" and "resinfo" message types. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4:0	Ignored
M0.2	31:17	Ignored
	16	**Force LOD to Zero:** If this bit is enabled, the calculated LOD is replaced with zero. The LOD is replaced just before entering the pseudocode in section 4.2.1.5, therefore the LOD is still subject to bias, overriding by sample_l delivered LOD, and clamping. Format = Enable
	15	**Alpha Write Channel Mask:** Enables the alpha channel to be written back to the originating thread. 0 = Alpha channel will be written back 1 = Alpha channel will not be written back **Programming Notes:** • a message with all four channels masked is not allowed. • this field is ignored for the sample_unorm*. The write channel mask is generated from the message type itself. • this field is ignored for the deinterlace message. • this field must be set to zero for sample_8x8 in VSA mode.
	14	**Blue Write Channel Mask:** See Alpha Write Channel Mask
	13	**Green Write Channel Mask:** See Alpha Write Channel Mask
	12	**Red Write Channel Mask:** See Alpha Write Channel Mask
	11:8	**U Offset:** the u offset from the _aoffimmi modifier on the "sample" or "ld" instruction. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages

DWord	Bit	Description
	7:4	**V Offset:** the v offset from the _aoffimmi modifier on the "sample" or "ld" instruction. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages
	3:0	**R Offset:** the r offset from the _aoffimmi modifier on the "sample" or "ld" instruction. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages
M0.1	31:0	Ignored
M0.0	31:0	Ignored

4.8.1.3 Payload Parameter Definition

The table below shows 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 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 not included as part of the message. The response lengths given below assume all channels are unmasked. SIMD16 messages with masked channels will have reduced response length.

message type	Message length	Response length	parameters												SIMD mode	API shader instruction
			0	1	2	3	4	5	6	7	8	9	10	1 1		
00	3	8	u												SIMD16	sample
00	5	8	u	v											SIMD16	sample
00	7	8	u	v	r										SIMD16	sample
00	4	4	u	v	r										SIMD8	sample
01	4	5	u	v	r										SIMD8	sample+killpix
00	9	8	u	v	r	bias									SIMD16	sample_b
01	9	8	u	v	r	lod									SIMD16	sample_l
01	2	1	u	v	r	lod									SIMD4x2	sample_l
10	9	8	u	v	r	ref									SIMD16	sample_c
00	2	1	u	v	r	ref									SIMD4x2	sample_c
00	6	4	u	v	r	bias	ref								SIMD8	sample_b_c
01	6	4	u	v	r	lod	ref								SIMD8	sample_l_c
01	3	1	u	v	r	lod	ref								SIMD4x2	sample_l_c
11	3	8	u												SIMD16	ld
11	5	8	u	v											SIMD16	ld
11	7	8	u	v	r										SIMD16	ld
11	4	4	u	v	r										SIMD8	ld
11	9	8	u	v	r	lod									SIMD16	ld
11	2	1	u	v	r	lod									SIMD4x2	ld
10	7	4	u	v	dudx	dvdx	dudy	dvdy							SIMD8	sample_g
10	10	4	u	v	r	dudx	dvdx	drdx	dudy	dvdy	drdy				SIMD8	sample_g
10	4	1	u	v	r		dudx	dvdx	drdx		dudy	dvdy	drdy		SIMD4x2	sample_g
10	3	8	lod												SIMD16	resinfo
10	2	1				lod									SIMD4x2	resinfo

Note that the SIMD8 messages actually contain only eight pixels of data. For the sample_g messages, this is due to the message length constraint of 16 registers not allowing these messages of 16 pixels. The Jitter will need to send two messages to the sampler to get 16 pixels of data.

4.8.1.4 Message Types

The behavior of each message type is as follows:

Message Type	Description
sample	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. **Programming Notes:** • 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 or any UINT or SINT format. • sample is not supported in SIMD4x2 mode.
sample+killpix	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. **Programming Notes:** • 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 or any UINT or SINT format. • sample+killpix is supported only in SIMD8 mode.
sample_b	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels, then the value in the "bias" parameter is added to the LOD for each pixel. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. 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. **Programming Notes:** • 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 or any UINT or SINT format. • sample_b is not supported in SIMD4x2 mode.
sample_l	The surface is sampled using the indicated sampler state. LOD is not computed, but instead is taken from the "lod" parameter. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. **Programming Notes:** • 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 a UINT or SINT format.

Message Type	Description
sample_c	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 WGF sample_c_lz instruction is implemented by issuing the sample_c message with **Force LOD to Zero** enabled in the message header or by issuing the sample_l_c message with the LOD parameter set to zero. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. • 1D and 2D arrays are not supported (**Depth** of the associated surface must be 0). • 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 is not allowed, as it is not possible for the hardware to compute LOD for SIMD4x2 messages. • Use of sample_c with SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, A32_FLOAT. • **[DevBW, DevCL] Errata:** When sample_c is used on a texture map with A16_FLOAT surface format, any value read in from the texture map that is a NaN will be treated like a + inf.
sample_b_c	This is a combination of sample_b and sample_c. Both the LOD bias and reference values are delivered. All restrictions applying to both sample_b and sample_c must be honored.
sample_l_c	This is a combination of sample_l and sample_c. Both the LOD and reference values are delivered. All restrictions applying to both sample_l and sample_c must be honored. However, unlike sample_c, sample_l_c *is* allowed as a SIMD4x2 message. **Programming Notes:** • [**DevBW, DevCL] Errata: SIMD4x2 sample_l_c** is not allowed and must be worked around using **SIMD8 sample_l_c.**
sample_g (sample_d)	The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The r coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The r coordinate contains the faceid, and the r gradients are ignored by hardware. **Programming Notes:** • 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 or any UINT or SINT format.

Message Type	Description
resinfo	The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned in the red, green, blue, and alpha channels respectively (UINT32 format). The width, height, and depth are 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 **Sampler State Pointer** and **Sampler Index** are ignored. **Programming Notes:** • **[DevBW-A,B] Errata:** if lod is > 0xf it must be forced to 0xf.
ld (includes ld2dms)	The surface is sampled using a default sampler state, indicated below. The "lod" parameter contains the LOD of the mip map to be sampled. The v and r channel may also 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 "lod" parameter. The **Sampler State Pointer** and **Sampler Index** are ignored. For the *ld* message type, 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 will return a value of one in the alpha channel) **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER. • **[DevBW-A,B] Errata:** Only non-array (**Depth** = 0) SURFTYPE_1D and SURFTYPE_2D are supported with "ld". • The **Surface Format** of the associated surface cannot be MONO8. • **[DevBW, DevCL] Errata:** For ld with SURFTYPE_BUFFER the lod channel MBZ. • **Errata:** Surface formats with 8 bits per channel and no alpha channel will return zero in the alpha channel.

Programming Notes:

- 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.

4.8.1.5 Parameter Types

sample*, LOD, and load4 messages

For all of the sample*, LOD, and load4 message types, all parameters are 32-bit floating point. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Normalized values range from [0,1] across the surface, with values outside the surface behaving as specified by the **Address Control Mode** in that dimension. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension, with values outside the surface being clamped to the surface.

Surface Type	u	v	r	ai
SURFTYPE_1D	normalized 'x' coordinate	unnormalized array index	ignored	ignored
SURFTYPE_2D	normalized 'x' coordinate	normalized 'y' coordinate	unnormalized array index	ignored
SURFTYPE_3D	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	ignored
SURFTYPE_CUBE	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	unnormalized array index

ld messages

For the ld message types, all parameters are 32-bit signed integers. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension. Input of any value outside of the range returns zero.

Surface Type	U	v	r
SURFTYPE_1D	unnormalized 'x' coordinate	unnormalized array index	ignored
SURFTYPE_2D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized array index
SURFTYPE_3D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized 'z' coordinate
SURFTYPE_BUFFER	unnormalized 'x' coordinate	ignored	ignored

4.8.1.6 SIMD16 Payload

The payload of a SIMD16 message provides addresses for the sampling engine to process 16 entities (examples of an entity are vertex and pixel). The number of parameters required to sample the surface depends on the state that the sampler/surface is in. Each parameter takes two message registers, with 8 entities, each a 32-bit floating point value, being placed in each register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this. For example, a 2D map using "sample_b" needs only u, v, and bias, but must send the r parameter as well.

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in section 4.8.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2.7	31:0	**Subspan 3, Pixel 3 (lower right) Parameter 0**
M2.6	31:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**
M2.5	31:0	**Subspan 3, Pixel 1 (upper right) Parameter 0**
M2.4	31:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**
M2.3	31:0	**Subspan 2, Pixel 3 (lower right) Parameter 0**
M2.2	31:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M2.1	31:0	**Subspan 2, Pixel 1 (upper right) Parameter 0**
M2.0	31:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M3 – Mn		Repeat packets 1 and 2 to cover all required parameters

4.8.1.7 SIMD8 Payload

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

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in section 4.8.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 – Mn		Repeat packet 1 to cover all required parameters

4.8.1.8 SIMD4x2 Payload

DWord	Bit	Description
M1.7	31:0	**Sample 1 Parameter 3** Specifies the value of the pixel's parameter 3. The actual parameter that maps to parameter 3 is given in the table in section 4.8.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Sample 1 Parameter 2**
M1.5	31:0	**Sample 1 Parameter 1**
M1.4	31:0	**Sample 1 Parameter 0**
M1.3	31:0	**Sample 0 Parameter 3**
M1.2	31:0	**Sample 0 Parameter 2**
M1.1	31:0	**Sample 0 Parameter 1**
M1.0	31:0	**Sample 0 Parameter 0**
M2		Parameters 4-7 if present
M3		Parameters 8-11 if present

4.8.2 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 or SIMD4x2).

4.8.2.1 SIMD16

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3.   The pixels written within each destination register is determined by the execution mask on the "send" instruction.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1.7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
W1.6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
W1.5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
W1.4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
W1.3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
W1.2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
W1.1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
W1.0	31:0	**Supspan 2, Pixel 0 (upper left) Red**
W2		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W3		**Subspans 3 and 2 of Green:** See W1 definition for pixel locations
W4		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W5		**Subspans 3 and 2 of Blue:** See W1 definition for pixel locations
W6		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W7		**Subspans 3 and 2 of Alpha:** See W1 definition for pixel locations

4.8.2.2 SIMD8

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.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written) **:** W4 is only delivered for the sample+killpix message type
W4.0	31:16	**Dispatch Pixel Mask:** This field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread.
	15:0	**Active Pixel Mask:** This field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. Since the SIMD8 message applies to only 8 pixels, only the low 8 bits within this field are used. The high 8 bits are always set to 1. **[DevBW, DevCL] Errata**: Active Pixel Mask needs to be ORed with the inverse of the EMask before it is ANDed with the DMask. Also if the sample instruction is within a conditional then the active pixel mask will be overwritten with the partial mask on each different sample instruction so this will have to be done for each instance of the sample instruction not just as the end.

4.8.2.3 SIMD4x2

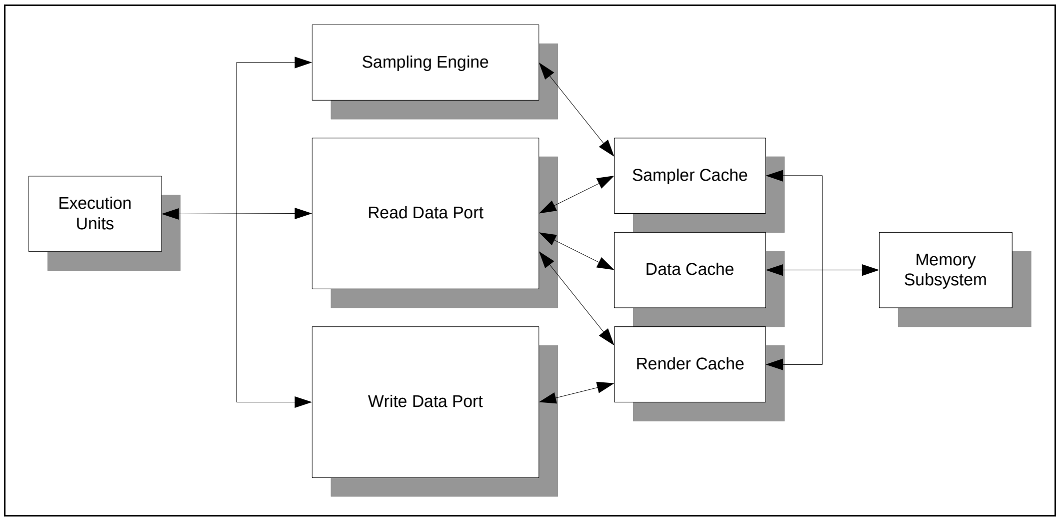
A SIMD4x2 writeback message always consists of a single message register containing all four channels of each of the two "pixels" (called "samples" here, as they are not really pixels) of data. The write channel mask bits as well as the execution mask on the "send" instruction are used to determine which of the channels in the destination register are overwritten. If any of the four execution mask bits for a sample is asserted, that sample is considered to be active. The active channels in the write channel mask will be written in the destination register for that sample. If the sample is inactive (all four execution mask bits deasserted), none of the channels for that sample will be written in the destination register.

DWord	Bit	Description
W0.7	31:0	**Sample 1 Alpha:** Specifies the value of the pixel's alpha channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Sample 1 Blue**
W0.5	31:0	**Sample 1 Green**
W0.4	31:0	**Sample 1 Red**
W0.3	31:0	**Sample 0 Alpha**
W0.2	31:0	**Sample 0 Blue**
W0.1	31:0	**Sample 0 Green**
W0.0	31:0	**Sample 0 Red**

5 Data Port

The Data Port provides all memory accesses for the Gen4 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.

The diagram below shows the two parts of the Data Port (Read and Write) 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 read and write data ports are considered to be separate shared functions, each with its own shared function identifier.

5.1 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 **Target Cache** field of the read data port message descriptor. The write data port message descriptor does not have an equivalent field as it only supports writes to the render cache.

5.1.1 Render Cache

The render cache is the only cache that supports both reads and writes. All writes must use this cache. In addition, all reads to a surface that is also being written should use this cache to avoid expensive flushing that would be required for coherency. The render cache supports both linear and tiled memory.

The render cache is intended to be used for the following surfaces:

- 3D render target surfaces
- destination surfaces for media applications
- intermediate working surfaces for media applications
- scratch space buffers

5.1.2 Data Cache

The data cache is a small, read-only cache that supports only linear memory. For 3D graphics, it is intended to be used only for constant buffers. For media and other generic applications, it may be used to load kernel constants such as filter coefficients as well as other linear data buffers such as compressed data buffer for HWMC.

In the hardware implementation on all of these devices, the data cache does not exist as a separate physical cache. It is mapped in hardware to the sampler cache.

5.1.3 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.

5.2 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.

5.2.1 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.

5.2.2 Stateless Model

The stateless model is used when a **Binding Table Index** (specified in the message descriptor) of 255 is specified. 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**
- Buffer Size = checked only against **General State Access Upper Bound**
- Surface Pitch = 16 bytes
- Utilize Fence = false
- Tiled = false

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

[DevBW-A,B] Erratum BWT006: Issuing a stateless access is UNDEFINED unless the **DAP Stateless Access ECO** bit in the SVG-Debug Workaround Control register is set. This is a DEBUG ONLY mode that lacks necessary security checks.

5.3 Write Commit

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

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

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

```
send r12 m1 DPWRITE ; issue write to render cache
mov m1 r3          ; assemble read message
mov r12 r12         ; block on write commit
send r13 m1 DPREAD  ; read same location as write
```

5.4 Read / Write Ordering

Hardware does not guarantee ordering between read and write messages issued to the data port, even between messages issued by the same thread. If ordering is important, software must guarantee ordering. For a write followed by a read to the same location, the write must use a write commit, and wait for the write commit to return before issuing the read message. For a read followed by a write to the same location, software must wait for the read data to be returned before issuing the write message.

5.5 Accessing Buffers

There are three data port messages used to access buffers. They are used for both constant buffers and scratch space buffers. All of these messages support only buffers, and can use the surface state model as well as the stateless model.

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

Message	Applications
OWord Block Read/Write	• 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

These messages ignore the surface format field of the state and perform no format conversion. The data contained in each channel is still not converted in any way.

5.6 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.

5.6.1 Boundary Behavior

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

Media Boundary Pixel Mode	Vertical Line Stride	Vertical Line Stride Offset	Usage Model	Behavior
0	0	X	normal frame	**0000**0123**3333**
0	1	0	normal field even	**0000**02**222222**
0	1	1	normal field odd	**1111**13**333333**
2	0	X	frame / progressive	**0000**0123**3333**
2	1	0	field even / progressive	**0000**02**333333**
2	1	1	field odd / progressive	**0000**13**333333**
3	0	X	frame / interlaced	**0101**0123**2323**
3	1	0	field even / interlaced	**0000**02**222222**
3	1	1	field odd / interlaced	**1111**13**333333**

5.7 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.

5.7.1 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 **Pixel Scoreboard Clear** bit is set in the message descriptor.

5.7.2 Dual Source [DevCL-B]

Note: Dual Source messages are not supported in DevBW and DevCL-A devices.

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 will clear the corresponding pixel scoreboard bits if the **Pixel Scoreboard Clear** bit in the message descriptor is set.

It is UNDEFINED to utilize a DualSource RT Write message when **Color Buffer Blend Enable** is DISABLED.

5.7.3 Replicate Data

The replicate data render target message is intended to be 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 **Pixel Scoreboard Clear** bit is set in the message descriptor.

5.7.4 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.

5.8 Flushing the Render Cache

A message that allows flushing the render cache is available for applications or for debug purposes. This message should not be used in normal 3D shaders, the render cache flushing mechanisms via PIPE_CONTROL or MI_FLUSH should be used instead as the render cache generally needs to be flushed on a level more global than that provided by a shader.

5.9 State

5.9.1 BINDING_TABLE_STATE

The data port uses the binding table to retrieve surface state. Refer to *Sampling Engine* for the definition of this state.

5.9.2 SURFACE_STATE

The data port uses the surface state for constant buffers, render targets, and media surfaces. Refer to *Sampling Engine* for the definition of this state.

5.10 Messages

5.10.1 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 two separate targets, **Data Port Read** and **Data Port Write**, each with its own target unit ID. Each target has its own set of message type encodings as shown below.

Restrictions:

- **[DevBW-A,B,C0, DevCL-A0] Errata:** use of any Sampling Engine message in the same workload (between pipeline flushes) with any Data Port read messages utilizing the Sampler Cache is not allowed.
- 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.

5.10.1.1 Message Descriptor

Bit	Description		
DATA PORT READ TARGET		**DATA PORT WRITE TARGET**	
15:14	Target Cache 00 = Data Cache 01 = Render Cache 10 = Sampler Cache 11 = Reserved	15	**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 5.3 for more details. Format = Enable
13:12	Read Message Type 00 = OWord Block Read 01 = OWord Dual Block Read 10 = Media Block Read 11 = DWord Scattered Read	14:12	Write Message Type 000 = OWord Block Write 001 = OWord Dual Block Write 010 = Media Block Write 011 = DWord Scattered Write 100 = Render Target Write 111 = Flush Render Cache All other encodings are reserved.
11:8	**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. Refer to section 5.2.2 for details on the stateless model. **Programming Notes:** • **[DevBW-A,B] Erratum BWT006**: Using a binding table index of 255 is UNDEFINED unless the **DAP Stateless Access ECO** bit in the SVG-Debug Workaround Control register is set. This is a DEBUG ONLY mode that lacks necessary security checks. Format = U8 Range = [0,255]		

5.10.1.2 Message Header

This header applies to the following data port messages:

- OWord Block Read/Write
- Unaligned OWord Block Read
- OWord Dual Block Read/Write
- DWord Scattered Read/Write

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

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:10	**Immediate Buffer Base Address.** Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Ignored
	7:0	**Dispatch ID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	**Global Offset.** Specifies the global byte offset into the buffer. • For the OWord messages, this offset must be OWord aligned (bits 3:0 MBZ) • For the DWord messages, this offset must be DWord aligned (bits 1:0 MBZ) Format = U32 Range = [0,FFFFFFF0h] for OWord messages Range = [0,FFFFFFFCh] for DWord messages
M0.1	31:0	Ignored
M0.0	31:0	Ignored

5.10.1.3 Write Commit Writeback Message

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

DWord	Bit	Description
W0.7:0		Reserved

5.10.2 OWord Block Read / Write

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

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be OWord aligned
- the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
- the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

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

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

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, is used as the single group of four with behavior following that in the preceding paragraph. **[DevBW], [DevCL] Errata:** Execution mask bits outside of those corresponding to the OWord being read/written cannot be asserted.

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 will not modify memory contents.

5.10.2.1 Message Descriptor

Bit	Description
12	Ignored
11	this bit is part of the **Read Message Type** field for the read version of this message)
10:8	**Block Size.** Specifies the number of contiguous OWords to be read or written 000 = 1 OWord, read into or written from the low 128 bits of the destination register 001 = 1 OWord, read into or written from the high 128 bits of the destination register 010 = 2 OWords 011 = 4 OWords 100 = 8 OWords 101 = 6 OWords all other encodings are reserved. **Programming Notes:** • The 6 OWord block size is valid only with **Data Port Constant Cache**.

5.10.2.2 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 (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.2.3 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 (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.3 OWord Dual Block Read / Write

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.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be OWord aligned
- the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
- the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

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

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

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

5.10.3.1 Message Descriptor

Bit	Description
12	Ignored
11:10	bit 11 is part of the **Read Message Type** field for the read version of this message)
9:8	**Block Size:** Specifies the number of OWords in each block to be read or written 00 = 1 OWord 10 = 4 OWords all other encodings are reserved.

5.10.3.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored

M1.4	31:0	**Block Offset 1.**
		Specifies the byte offset of OWord Block 1 into the surface. Must be OWord aligned (bits 3:0 MBZ).
		Format = U32
		Range = [0,FFFFFFF0h]

M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	**Block Offset 0**

5.10.3.3 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 (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.3.4 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 (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.4 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:

- the only surface type allowed is SURFTYPE_2D. Because of this, the stateless surface model is not supported with this message.
- the surface format is used to determine the pixel structure for boundary clamp, the raw data from the surface is returned to the thread without any format conversion nor filtering operation
- the target cache cannot be the data cache
- the surface base address must be 32-byte aligned
- When a surface is XMajor tiled, (**tile walk** field in the surface state is set to TILEWALK_XMAJOR), a memory area mapped through the Render Cache cannot be read and/or wrote 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 GRF are uncertain.
- The block width and offset should be aligned to the size of pixels stored in the surface. For a surface with 8bpp pixels for example, the block width and offset can be byte aligned. For a surface with 16bpp pixels, it is word aligned.
  - For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. dword aligned).
- The write form of message has the additional restriction that both **X Offset** and **Block Width** must be DWord aligned.

- The read form of message also has the additional restriction that both **X Offset** and **Block Width** must be DWord aligned.
- **[DevBW-A] Erratum BWT001**: Surfaces being *read* with this message by the render cache must be tiled. Writes to linear surfaces are allowed.
- **[DevBW-A] Erratum**: A memory area mapped through the Render Cache cannot be read and/or written in mixed frame and field modes.

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 of B0B0B0B0, and that for 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 of B0B1B0B1, and that for right boundary is B2B3B2B3.
  - This rule applies to all surface formats with BPE of 16. As the data port does not perform format conversion, only the formats with integer data types may be useful in practice.
- For special surfaces with 16bpp pixels YUV422 packed format, there are two basic cases depending on the Y location: YUYV (surface format YCRCB_NORMAL) and UYVY (surface format YCRCB_SWAPY). Boundary handling for YVYU (surface format YCRCB_SWAPUV) is the same as that for YUYV. Similarly, boundary handling for VYUY (surface format YCRCB_SWAPUVY) is the same as that for UYVY. Note that these four surface formats have 16bpp pixels, even though the BPE fields are set to zero according to the table in the Surface Formats Section.
  - For a boundary dword Y0U0Y1V0, to replicate the left boundary, we get Y0U0**Y0**V0, and to replicate the right boundary, we get **Y1**U0Y1V0.
  - For a boundary dword U0Y0V0Y1, to replicate the left boundary, we get U0Y0V0**Y0**, and to replicate the right boundary, we get U0**Y1**V0Y1.
- For a surface with 32bpp pixels, the boundary dword pixel is replicated.
  - This rule applies to all surface formats with BPE of 32. As the data port does not perform format conversion, some of the formats may not be useful in practice.

Hardware behavior for any other surface types is undefined.

5.10.4.1 Message Descriptor

Bit	Description
12	**Cache Allocation Method** This field is only allowed to be 1 only if resulting Vertical Line Stride (from surface state or being overridden by this message) is 1. This field is only valid for Sampler Cache read messages. This field is ignored for Render Cache messages (read/write). 0 = frame cache lines 1 = field cache lines this bit is part of the **Message** Type fields
11	Pixel Scoreboard Clear. Reserved : MBZ
10	**Vertical Line Stride Override** Specifies whether the **Vertical Line Stride** and **Vertical Line Stride Offset** fields in the surface state should be replaced by bits 9 and 8 below. If this field is 1, **Height** in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:

Vertical Line Stride (in surface state)	Override Vertical Line Stride	Derived 1-based surface height (As a function of the 0-based Height in surface state)
0	0	**Height** + 1 (Normal)
0	1	(**Height** +1) / 2 *Restriction: (Height + 1) must be an even number.*
1	0	(**Height** + 1) * 2
1	1	**Height** + 1 (Normal)

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, both Override Vertical Line Stride and Override Vertical Line Stride Offset will be set to 1, then the derived surface height (of the field) will be 240 ((Height + 1) / 2). In contrary, 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 will use the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset set to 0) will result 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**

[DevBW-A] Erratum: This field is ignored by hardware.

Bit	Description
9	**Override Vertical Line Stride** Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures. Format = U1 in lines to skip between logically adjacent lines **[DevBW-A] Erratum**: This field is ignored by hardware.
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) **[DevBW-A] Erratum:** This field is ignored by hardware.

5.10.4.2 Message Header

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:8	Ignored
	7:0	**FFTID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:5	Ignored
	4:3	Ignored
	3:2	Ignored
	1	Ignored
	0	Ignored
M0.2	31:22	Ignored

DWord	Bit	Description
	21:16	**Block Height.** Height in rows of block being accessed. **Programming Notes:** • The Block Height is restricted to the following maximum values depending on the Block Width:

Block Width (bytes)	Maximum Block Height (rows)
1-4	64
5-8	32
9-16	16
17-32	8

DWord	Bit	Description

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

	15:5	Ignored
	4:0	**Block Width.** Width in bytes of the block being accessed. **Programming Notes:** • Must be DWord aligned for the write form of the message. • This field must also be DWord aligned for the read form of the message. Format = U5 Range = [0,31] representing 1 to 32 Bytes
M0.1	31:0	**Y offset.** The Y offset of the upper left corner of the block into the surface. Format = S31
M0.0	31:0	**X offset.** 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 **X offset** field defines the offset in the input message block. This may differ from the offset in the surface if Color Processing is enabled due to format conversion. This field must also be DWord aligned for the read form of the message. Format = S31

5.10.4.3 Message Payload (Write)

DWord	Bit	Description
M1:n		**Write Data.** The format of the write data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**.

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.

Message Format	**31:24**	**23:16**	**15:8**	**7:0**
YUV 4:2:2, 8 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 8 bits per channel	Alpha (A)	luminance (Y)	Cb (U)	Cr (V)
Message Format	**63:48**	**47:32**	**31:16**	**15:0**
YUV 4:2:2, 16 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 16 bits per channel	Alpha (A)	Cr (V)	luminance (Y)	Cb (U)

5.10.4.4 Writeback Message (Read)

DWord	Bit	Description
W0:n		**Read Data.** The format of the read data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**.

5.10.5 DWord Scattered Read/Write

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.

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.

Hardware does not check for or optimize for cases where offsets are equal or contiguous, thus for optimal performance in these cases a different message may provide higher performance.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be DWord 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:

- 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.

5.10.5.1 Message Descriptor

Bit	Description
12	Ignored
11:10	bit 11 is part of the **Read Message Type** field for the read version of this message)
9:8	**Block Size.** Specifies the number of DWords to be read or written 10 = 8 DWords 11 = 16 DWords All other encodings are reserved.

5.10.5.2 Message Payload

DWord	Bit	Description

M1.7	31:0	**Offset 7.**
		Specifies the byte offset of DWord 7 into the surface. Must be DWord aligned (bits 1:0 MBZ).
		Format = U32
		Range = [0,FFFFFFFCh]

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**

5.10.5.3 Additional Message Payload (Write)

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

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords (discard low 2 bits). The **DWord** array index is also in units of DWords.

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

5.10.5.4 Writeback Message (Read)

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

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords (discard low 2 bits). The **DWord** array index is also in units of DWords.

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

5.10.6 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 will also be 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.

Restrictions:

- All surface types are allowed.
- Dual Source messages are not supported on **DevBW** and **DevCL-A**
- 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.
- Only one pair of dual source messages is allowed per thread, as these messages implicitly clear the pixel scoreboard. In addition, a thread sending dual source messages is not allowed to send any other render target write messages.
- 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 will cause undefined behavior.
- The dual source message cannot be used if the **Antialias Alpha Present to Render Target** bit in the message header is enabled.
- The dual source message cannot be used if the **Alpha Test Enable** bit in COLOR_CALC_STATE is enabled.
- This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1)

Execution Mask. 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.

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

5.10.6.1 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.

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)

Message Size	Num Samples	Slot Mapping
SIMD16	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]
SIMD8	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]

5.10.6.2 Message Descriptor

Bit	Description
11	**Last Render Target Select.** This bit must be set on the last render target write message sent for each group of pixels. For single render target pixel shaders, this bit is set on all render target write messages. For multiple render target pixel shaders, this bit is set only on messages sent to the last render target.
10:8	**Message Type.** This field specifies the type of render target message. For the dual source messages, the low bit indicates which subspan channels to use for the X/Y addresses, stencil, and antialias alpha data. **Programming Notes:** • Replicated data (**Message Type** = 001) is only supported when accessing tiled memory. Using this Message Type to access linear (untiled) memory is UNDEFINED. • **[DevBW, DevCL-A] Errata:** Dual Source messages are not supported • **[DevCL-B]:** The SIMD8 dual source message using subspan 2 & 3 slots (encoding 011) is not supported 000 = SIMD16 single source message 001 = SIMD16 single source message with replicated data 010 = SIMD8 dual source message, use subspan 0 & 1 slots 011 = SIMD8 dual source message, use subspan 2 & 3 slots 100 = SIMD8 single source message, use subspan 0 & 1 slots 101-111: Reserved

5.10.6.3 Message Header

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

DWord	Bit	Description
M0.7	31:0	Debug
M0.6	31:0	Debug
M0.5	31:8	Ignored
	7:0	**FFTID.** The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:6	**Color Calculator State Pointer.** Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:6]
	5:0	Ignored
M0.0	31:16	**Dispatched Pixel Enables.** One bit per pixel indicating which pixels were originally enabled when the thread was dispatched. This field is only required for the end-of-thread message and on all dual-source messages. The **Dispatched Pixel Enables** *must be unmodified* from the ones sent when the pixel shader thread was initiated. If the **Dispatched Pixel Enables** are modified, behavior is undefined.
	15:0	**Pixel Enables.** One bit per pixel indicating which pixels are still lit based on kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer.
M1.7	31	Ignored

	30:27	**Viewport Index.** Specifies the index of the viewport currently being used.
		
		Format = U4
		
		Range = [0,15]

DWord	Bit	Description
	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. **face** / **Render Target Array Index**: x — 0 -x — 1 y — 2 -y — 3 z — 4 -z — 5 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.
	15:0	Ignored
M1.6	31	**Front/Back Facing Polygon.** Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0 = Front Facing 1 = Back Facing
	30	Ignored
	29	**Source Depth Present to Render Target.** Indicates that source depth is included in the message. If **Destination Depth Present** is also set, the depth test and conditional write of the depth buffer must be performed. If **Destination Depth Present** is not set, no depth test is performed but the source depth value is conditionally written to the depth buffer.
	28	**Destination Depth Present to Render Target.** Indicates that destination depth is included in the message, and that the depth test and conditional write of the depth buffer must be performed. It is not valid to have **Destination Depth Present** without **Source Depth Present**.
	27	**Destination Stencil Present to Render Target.** Indicates that destination stencil is included in the message, and that the stencil test and conditional write of the stencil buffer must be performed.
	26	**Antialias Alpha Present to Render Target.** Indicates that antialias alpha is included in the message, and that the antialias function must be performed.
	25:0	Ignored

DWord	Bit	Description
M1.5	31:16	**Y3.** Y coordinate for upper-left pixel of subspan 3 Format = U16
	15:0	**X3.** X coordinate for upper-left pixel of subspan 3 Format = U16
M1.4	31:16	**Y2**
	15:0	**X2**
M1.3	31:16	**Y1**
	15:0	**X1**
M1.2	31:16	**Y0**
	15:0	**X0**
M1.1	31:0	Ignored
M1.0	31:0	Ignored

5.10.6.4 Stencil and Antialias Alpha Payload

The stencil and antialias alpha registers, if included, appears as message register 2 (M2), immediately following the header.

Note that the Antialias Alpha values are U0.4.

DWord	Bit	Description
M2.7	31:28	**Antialias Alpha for Subspan 3, Pixel 3 (lower right)** Format = U0.4 This register is only included if the **Antialias Alpha Present** or **Destination Stencil Present** bit is set.
	27:24	**Antialias Alpha for Subspan 3, Pixel 2 (lower left)**
	23:20	**Antialias Alpha for Subspan 3, Pixel 1 (upper right)**
	19:16	**Antialias Alpha for Subspan 3, Pixel 0 (upper left)**
	15:12	**Antialias Alpha for Subspan 2, Pixel 3 (lower right)**
	11:8	**Antialias Alpha for Subspan 2, Pixel 2 (lower left)**
	7:4	**Antialias Alpha for Subspan 2, Pixel 1 (upper right)**
	3:0	**Antialias Alpha for Subspan 2, Pixel 0 (upper left)**
M2.6	31:28	**Antialias Alpha for Subspan 1, Pixel 3 (lower right)**
	27:24	**Antialias Alpha for Subspan 1, Pixel 2 (lower left)**
	23:20	**Antialias Alpha for Subspan 1, Pixel 1 (upper right)**
	19:16	**Antialias Alpha for Subspan 1, Pixel 0 (upper left)**
	15:12	**Antialias Alpha for Subspan 0, Pixel 3 (lower right)**
	11:8	**Antialias Alpha for Subspan 0, Pixel 2 (lower left)**
	7:4	**Antialias Alpha for Subspan 0, Pixel 1 (upper right)**

DWord	Bit	Description
	3:0	**Antialias Alpha for Subspan 0, Pixel 0 (upper left)**
M2.5:4		Reserved
M2.3	31:24	**Destination Stencil for Subspan 3, Pixel 3 (lower right)** Format = U8
	23:16	**Destination Stencil for Subspan 3, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 3, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 3, Pixel 0 (upper left)**
M2.2	31:24	**Destination Stencil for Subspan 2, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 2, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 2, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 2, Pixel 0 (upper left)**
M2.1	31:24	**Destination Stencil for Subspan 1, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 1, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 1, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 1, Pixel 0 (upper left)**
M2.0	31:24	**Destination Stencil for Subspan 0, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 0, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 0, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 0, Pixel 0 (upper left)**

This payload is included if the Message Type is SIMD16 single source. The value of 'm' here is equal to 2 if both stencil and antialias alpha are not present, otherwise it is equal to 3.

DWord	Bit	Description
Mm.7	31:0	**Subspan 1, Pixel 3 (lower right) Red.** Specifies the value of the pixel's red channel. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
Mm.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
Mm.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
Mm.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
Mm.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
Mm.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
Mm.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
M(m+1)		**Subspans 1 and 0 of Green.** See Mm definition for pixel locations
M(m+2)		**Subspans 1 and 0 of Blue.** See Mm definition for pixel locations
M(m+3)		**Subspans 1 and 0 of Alpha** See Mm definition for pixel locations
M(m+4).7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
M(m+4).6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
M(m+4).5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
M(m+4).4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
M(m+4).3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
M(m+4).2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
M(m+4).1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
M(m+4).0	31:0	**Supspan 2, Pixel 0 (upper left) Red**
M(m+5)		**Subspans 3 and 2 of Green.** See M3 definition for pixel locations
M(m+6)		**Subspans 3 and 2 of Blue.** See M3 definition for pixel locations
M(m+7)		**Subspans 3 and 2 of Alpha.** See M3 definition for pixel locations

5.10.6.5 Color Payload: SIMD8 Single Source

This payload is included if the Message Type is SIMD8 single source. The value of 'm' here is equal to 2 if both stencil and antialias alpha are not present, otherwise it is equal to 3.

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

5.10.6.6 Color Payload: SIMD16 Replicated Data

This payload is included if the Message Type specifies single source message with replicated data. One set of R/G/B/A data is included in the message, and this data is replicated to all 16 pixels.

This message is legal with color data only. 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' here is equal to 2.

Programming Notes:

- This message is allowed only on tiled surfaces

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

5.10.6.7 Color Payload: SIMD8 Dual Source [DevCL-B]

This payload is included if the **Message Type** specifies dual source message. The value of 'm' here is equal to 2 if both stencil and antialias alpha are not present, otherwise it is equal to 3.

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

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

5.10.6.8 Depth Payload

The depth registers, if included, appear immediately following the color payload.

For the SIMD8 messages, only slot 7:0 data is sent, or only slot 15:8 depending on the **Message Type** encoding. Any complete message register containing ignored data cannot be delivered.

DWord	Bit	Description
Mn.7	31:0	**Source Depth for Slot 7** Format = IEEE_Float This and the next register is only included if **Source Depth Present** bit is set.
Mn.6	31:0	**Source Depth for Slot 6**
Mn.5	31:0	**Source Depth for Slot 5**
Mn.4	31:0	**Source Depth for Slot 4**
Mn.3	31:0	**Source Depth for Slot 3**
Mn.2	31:0	**Source Depth for Slot 2**
Mn.1	31:0	**Source Depth for Slot 1**
Mn.0	31:0	**Source Depth for Slot 0**
M(n+1).7	31:0	**Source Depth for Slot 15**
M(n+1).6	31:0	**Source Depth for Slot 14**
M(n+1).5	31:0	**Source Depth for Slot 13**
M(n+1).4	31:0	**Source Depth for Slot 12**
M(n+1).3	31:0	**Source Depth for Slot 11**
M(n+1).2	31:0	**Source Depth for Slot 10**
M(n+1).1	31:0	**Source Depth for Slot 9**
M(n+1).0	31:0	**Source Depth for Slot 8**
Mk.7	31:0	**Destination Depth for Slot 7** Format depends on depth buffer surface format. Software should not modify the destination depth fields from what was delivered in the thread payload. This and the next register is only included if **Destination Depth Present** bit is set.
Mk.6	31:0	**Destination Depth for Slot 6**
Mk.5	31:0	**Destination Depth for Slot 5**
Mk.4	31:0	**Destination Depth for Slot 4**
Mk.3	31:0	**Destination Depth for Slot 3**
Mk.2	31:0	**Destination Depth for Slot 2**
Mk.1	31:0	**Destination Depth for Slot 1**

DWord	Bit	Description
Mk.0	31:0	**Destination Depth for Slot 0**
M(k+1).7	31:0	**Destination Depth for Slot 15**
M(k+1).6	31:0	**Destination Depth for Slot 14**
M(k+1).5	31:0	**Destination Depth for Slot 13**
M(k+1).4	31:0	**Destination Depth for Slot 12**
M(k+1).3	31:0	**Destination Depth for Slot 11**
M(k+1).2	31:0	**Destination Depth for Slot 10**
M(k+1).1	31:0	**Destination Depth for Slot 9**
M(k+1).0	31:0	**Destination Depth for Slot 8**

5.10.6.9 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, thus they are not shown in the table. All cases not shown in this table are illegal.

Key:
s0, s1 = source 0, source 1
1/0 = subspan 1 & 0
3/2 = subspan 3 & 2
sZ = source depth
dZ = destination depth
sten = stencil & antialias alpha

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

5.10.7 Flush Render Cache

This message causes a flush of the render cache. The flush occurs in-order relative to message arrival at the write data port. It is not synchronized with messages to the read data port.

If the **Send Write Commit Message** bit in the message descriptor is set for this message, the writeback message is delivered after the cache flush has been completed.

5.10.7.1 Message Descriptor

Bit	Description
12	Ignored
11:8	Ignored

5.10.7.2 Message Payload

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5:0	31:0	Ignored

§§

6 Extended Math

The Extended Math (EM) shared function supports math functions not available in the GEN4 execution units. These math functions include reciprocal, logarithms, square root, integer divide, and transcendental functions, etc. EM does not use any state: all information needed to perform its operations is provided in the incoming messages.

The Extended Math operates on one data element per clock through a compute pipeline. A request message may contain multiple data elements. These data elements are put in the compute pipeline in series. Data elements from different request messages are also put in the pipeline. When the computation is completed for all data elements in a request message, output data are assembled for the request and sent back to the requesting thread as a writeback message. Many math functions require data to be processed through the compute pipeline in multiple passes. The throughput and latency for a given message depends on the math function type and some times depends on the input data values.

Unlike other shared functions in the GEN4 architecture, when a thread issues multiple requests to the Extended Math, EM may return the results of those requests out of order. Note that result register dependency makes this behavior transparent to the thread (except in the case where the thread manually manages post-destination register dependency).

Like other shared functions in the GEN4 architecture, EM does not guarantee any ordering between requests from different threads.

6.1 Messages

Restrictions:

- Use of any message to the Extended Math with the **End of Thread** bit set in the message descriptor is not allowed.
- The Extended Math supports vector operations up to 8 channels. It only looks at the lower 8 channel enables (execution mask bits), and ignores the higher 8.

6.1.1 Initiating Message

6.1.1.1 Message Descriptor

Bit	Description
19	This bit is not part of the shared function specific message descriptor**.**
18:9	Reserved : MBZ Bits 18:16 are not part of the shared function specific message descriptor.
8	**Snapshot bit.** When set to 1 the EM unit will latch debugging information for this message into a MMIO register. See the *Debugging* chapter for a description and layout of bits in the MMIO
7	**Source Structure.** This bit indicates whether the operation is based on vector inputs or scalar inputs. If this bit is not set, the Extended Math performs the indicated math function on a channel by channel basis. For an enabled channel, EM takes the input data from the corresponding channel and outputs the result in the same position. If this bit is set, EM performs the math function on a 4-channel group basis. If any of the 4 channels within a group is enabled, the data on the first channel (channel 0) is used as the input. The result is broadcasted to all enabled channels within the group. See section 6.1.1.2 below for more details. 0 = vector structure 1 = scalar structure
6	Saturate Control 0 = no saturate 1 = saturate result to [0,1] range (allowed only on floating point math functions)
5	**Precision.** This bit provides a hint whether the indicated math function is performed in full precision or partial precision. It is only valid for floating point math functions when the floating point mode is in alternative mode. It is ignored if the floating point mode is in IEEE754 mode. Floating point mode is selected via the **Floating Point Mode** bit in CR0. This bit is also ignored for integer math functions. See section ☐ for more details. 0 = use full precision 1 = use partial precision

Bit	Description
4	**Integer Type.** Determines the data type for both source and destination operands of the INT DIV functions. Ignored for other functions. 0 = unsigned integer 1 = signed integer
3:0	**Math Function.** For floating point math functions (1h to Ah), the floating point mode signal in the request message (originated from the Floating Point Mode bit in CR0) determines whether the operation is in IEEE754 floating point mode or in alternative floating point mode. Functions LOG and EXP are base 2. SIN, COS, SINCOS take inputs in radians. 0h: Reserved 1h: INV (reciprocal) 2h: LOG 3h: EXP 4h: SQRT 5h: RSQ 6h: SIN 7h: COS 8h: SINCOS 9h: Reserved Ah: POW Bh: INT DIV – return quotient and remainder Ch: INT DIV – return quotient only Dh: INT DIV – return remainder only Eh: Reserved Fh: Reserved

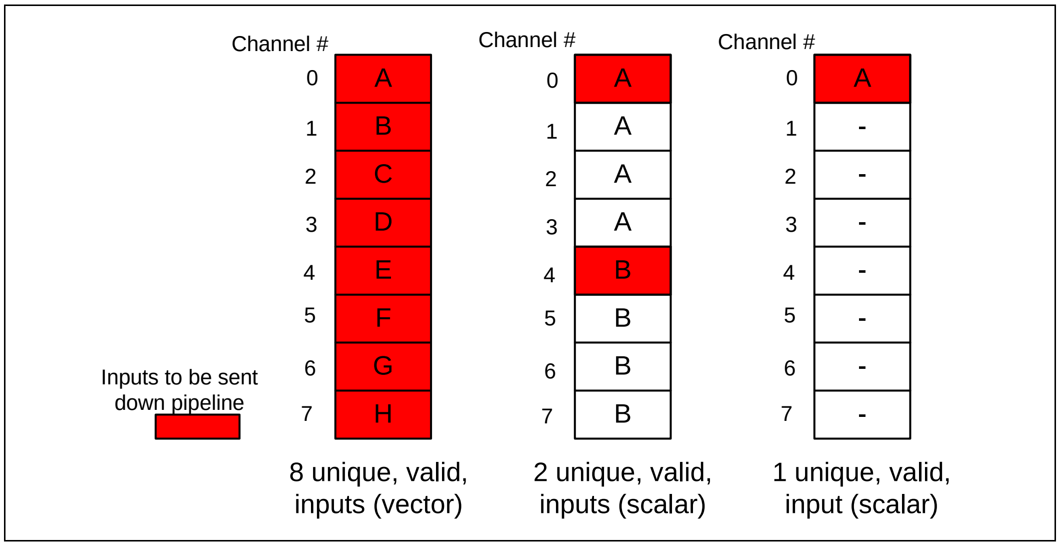
6.1.1.2 Scalar and Vector Mode

For a given request message, the Extended Math examines the 8-bit channel enable field and the Source Structure field in the message descriptor to determine which dwords contain valid inputs. There are two general cases that EM sees.

- **Vector mode**: The first case is when the Source Structure is a vector structure. In this vector mode, 8 input data channels contain 8 unique input values. The channel enable bits in the sideband determine which one of the 8 input values are valid and therefore need to be computed and outputted. It is possible that none of the channels are enabled, or all 8 channels are enabled, or anything in between. EM only sends the valid input values into the compute pipeline to achieve higher throughput. As the channel enable field is forwarded to the writeback message bus, only the resulting values with channel enable bit on are written back to the requesting thread's GRF register.
- **Scalar mode**: The second case is when the Source Structure is a scalar structure. In this scalar mode, there may be up to 2 unique input values present, one for each group of 4 channels. The 2 unique input values reside in the first channel of each group of 4, channel 0 and channel 4, specifically. The computed results of the two scalar inputs are replicated to the corresponding 4 channels. The sideband channel enable field determines which channels are enabled at the final output. It is obvious that as long as any bit out of a group of four channel-enable bits are set, the corresponding scalar data must be computed. Inversely, if all four channel

enable bits in a group are zero, computation of the corresponding scalar is skipped.

A subset of the scalar mode is when there is only one valid input. In this case the channel enable field will show that one of the two groups of four does not contain valid data. These three cases are illustrated below:


Channel #
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
Inputs to be sent down pipeline
8 unique, valid, inputs (vector)
Channel #
0 A
1 A
2 A
3 A
4 B
5 B
6 B
7 B
2 unique, valid, inputs (scalar)
Channel #
0 A
1 -
2 -
3 -
4 -
5 -
6 -
7 -
1 unique, valid, input (scalar)

6.1.1.3 Message Payload

All incoming messages are comprised of a single message register except the POW function and INT DIV, which consist of two message registers.

The lower 8 bits of the channel enables (execution mask) are used as the (dword) channel enables for the math function operation.

DWord	Bit	Description
M0.7	31:0	**Operand0[7].** The value of Operand0 for element 7 For the POW function, this operand is the base For the INT DIV functions, this operand is the denominator For all other functions, this operand is the single input operand Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
M0.6	31:0	**Operand0[6].** Refer to Operand0[7] above for the function of this operand.
M0.5	31:0	**Operand0[5].** Refer to Operand0[7] above for the function of this operand.
M0.4	31:0	**Operand0[4].** Refer to Operand0[7] above for the function of this operand.
M0.3	31:0	**Operand0[3].** Refer to Operand0[7] above for the function of this operand.
M0.2	31:0	**Operand0[2].** Refer to Operand0[7] above for the function of this operand.
M0.1	31:0	**Operand0[1].** Refer to Operand0[7] above for the function of this operand.
M0.0	31:0	**Operand0[0].** Refer to Operand0[7] above for the function of this operand.
M1.7	31:0	**Operand1[7].** The value of Operand1 for element 7 For the POW function, this operand is the power For the INT DIV functions, this operand is the numerator For all other functions, this data phase of the message is not present Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
M1.6	31:0	**Operand1[6].** Refer to Operand1[7] above for the function of this operand.
M1.5	31:0	**Operand1[5].** Refer to Operand1[7] above for the function of this operand.
M1.4	31:0	**Operand1[4].** Refer to Operand1[7] above for the function of this operand.
M1.3	31:0	**Operand1[3].** Refer to Operand1[7] above for the function of this operand.
M1.2	31:0	**Operand1[2].** Refer to Operand1[7] above for the function of this operand.
M1.1	31:0	**Operand1[1].** Refer to Operand1[7] above for the function of this operand.
M1.0	31:0	**Operand1[0].** Refer to Operand1[7] above for the function of this operand.

6.1.2 Writeback Message

Writeback messages for most EM functions contain a single GRF register. The exceptions to this rule are SINCOS and INT DIV. SINCOS returns two GRF registers, the first register contains the computed Sine of the inputs, and the second contains the computed Cosine values. INT DIV returns the quotient in the first GRF register and the remainder in the second GRF register. The two GRF registers are adjacent.

The lower 8 bits of the channel enables (execution mask) of the writeback bus are the same 8 (dword) channel enables of the request message. Because EM supports vector operations with a maximum of 8 channels, the higher 8 bits of the channel enables are set to 0. The same 16-bit channel enables are repeated for the second GRF register write, if present.

DWord	Bit	Description
W0.7	31:0	**Result0[7].** The value of Result0 for element 7 For the SINCOS function, this result is the sine For the INT DIV (return quotient and remainder) functions, this result is the quotient For all other functions, this result is the single output result Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
W0.6	31:0	**Result0[6]**
W0.5	31:0	**Result0[5]**
W0.4	31:0	**Result0[4]**
W0.3	31:0	**Result0[3]**
W0.2	31:0	**Result0[2]**
W0.1	31:0	**Result0[1]**
W0.0	31:0	**Result0[0]**
W1.7	31:0	**Result1[7].** The value of Result1 for element 7 For the SINCOS function, this result is the cosine For the INT DIV (return quotient and remainder) functions, this result is the remainder For all other functions, this data phase of the message is not present Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
W1.6	31:0	**Result1[6]**
W1.5	31:0	**Result1[5]**
W1.4	31:0	**Result1[4]**
W1.3	31:0	**Result1[3]**
W1.2	31:0	**Result1[2]**
W1.1	31:0	**Result1[1]**
W1.0	31:0	**Result1[0]**

6.2 Performance

The Extended Math shared function unit supports extended math functions with up to 8 data channels per request. Computations for a vector request are performed channel by channel on a serial execution pipeline. Most functions require iterative computations. For example, SQRT takes three rounds of computation in the serial execution pipeline. The latency for each round is about 22 clocks. Trigonometric functions may take variable number of rounds depending on the input data. For certain math functions, the throughput with partial precision computation in alternative floating point mode is higher than the full precision computation. After computations for all channels of a request are completed, data vectors (of one or two phases) are assembled before the writeback message is sent back to the requesting thread.

The following table shows the number of rounds per element for each function type. The table may be used to estimate the utilization of the extended math unit and the minimal latency of the message.

Function	Throughput (rounds/element)	Note
INV	1	
LOG	Partial: 2 Full: 3	Computes Log base 2
SQRT	3	Implemented as: $\sqrt{x} = x * 1/\sqrt{x}$
RSQ	2	
EXP	Full: 4 Partial: 3	Both partial and full precision versions have the same throughput. Computes 2^x (anti-log)
POW	8	
SIN	Min: 5 Max: 12 Typical: 6	Trigonometric functions are the only ones with variable throughput. Throughput depends on the input data range. Input is in radians
COS	Same as SIN	Input is in radians
SINCOS	See SIN	The two-output-phase SINCOS function is implemented as back to back SIN and COS functions. Input is in radians
INT DIV	Quotient: 3 Remainder: 4	

To best utilize the extended math shared function, programmers should consider the following characteristics of the shared function:

- In vector mode, only the enabled channels consume computation rounds, while the disabled channels do not.
- In scalar mode, one data element is computed for a group of 4 channels if any of the 4 channels is enabled. If all 4 channels are disabled, no compute cycle is wasted for the group.

6.3 Function Reference

A math function may take one request message register (src0) or two request message registers (src0 and src1), and may output one writeback message register (dst0) or two writeback message registers (dst0 and dst1).

Vector mode or scalar mode is determined by the Source Structure field of message descriptor.

The operations is based on the channel enables as noted by EMask.

6.3.1 INV

Description Computes reciprocal of src0 (32-bit float format) and stores computed result in dest as a 32-bit float

Format: INV <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
        dst0.channel[n] = 1 / src0.channel[srcCh]
    }
}
```

Precision: 1 ULP

Src->	+inf	+0 / +Denorm	- 0 / - Denorm	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN
Dest – ALT mode	▨	+FLT_MAX	-FLT_MAX	▨	NaN

6.3.2 LOG

Description: Computes Log_2 of Src0 and stores computed result in Dest. Both src0 and dest are 32-bit FP values

Format: LOG <dst0> <src0>

Pseudocode:
```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst0.channel[n] = $Log_2$(src0.channel[srcCh])
    }
}
```

Precision: +/- 2-21 max relative error – Full precision
+ / - 2-10 max relative error- partial precision

Notes: In ALT mode log is computed as Log_2 (abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**-inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**-FLT_MAX**	**-FLT_MAX**		**+F**	**NaN**

6.3.3 EXP

Description: Computes 2^{src0} and stores computed result in Dest. Both src0 and dest are 32-bit FP values

Format: EXP <dst0> <src0>

Pseudocode:
```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst0.channel[n] = $2^{src0.channel[srcCh]}$
        }
    }
```

Precision: + / - 2-21 max relative error – full precision
+/- 2-10 max relative error – partial precision

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**1**	**1**	**0**	**+F**	**NaN**
Dest – ALT mode	▨	**1**	**1**	▨	**+F**	**NaN**

6.3.4 SQRT

Description: Computes square-root of src0 and stores computed result in dest. Both src0 and dest are 32-bit FP values

Format: SQRT <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst0.channel[n] = $\sqrt{SRC0.\text{channel[srcCh]}}$
    }
}
```

Precision: 1 ULP

Notes: In ALT mode SQRT is computed as SQRT(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**0**	**-0**	**NaN**	**NaN**	**NaN**
Dest – ALT mode	▨	**0**	**0**	▨	**+F**	**NaN**

6.3.5 RSQ

Description: Computes reciprocal square-root of src0 and stores computed result in dest. Both src0 and dest are 32-bit FP values

Format: RSQ <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst.channel[n] = $1/\sqrt{SRC0.\text{channel}[n]}$
    }
}
```

Precision: 1 ULP

Notes: In ALT mode RSQ is computed as RSQ(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+0**	**+inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode	▨	**+FLT_MAX**	**+FLT_MAX**	▨	**+F**	**NaN**

6.3.6 POW

Description: Computes abs(src0) raised to the src1 power and stores computed result in dst0. Src0, src1, and dst0 are 32-bit FP values. Src1 is always scalar value.

Format: POW <dst0> <src0> <src1>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst0.channel[n] = $2^{src1 \cdot \log_2(abs(src0.\text{channel[srcCh]}))}$
    }
}
```

Precision: 2^-15 relative error

IEEE Mode:

Src0->

Src1	abs(F > 1)	abs (F < 1)	abs(+F = = 1)	+inf	+0 / +Denorm	-Denorm / -0	-inf	NaN
+inf	+inf	0	NaN	+inf	0	0	+inf	NaN
+0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F			+inf	0	0	NaN	NaN

ALT Mode:

Src0->

Src1	+F	+	+0 / +Denorm	-0 / -	-	-F	NaN
+inf							
+0 / Denorm	1		1	1		1	NaN
-0 / Denorm	1		1	1		1	NaN
-inf							
-F	+F		+FLT_MAX	+FLT_MAX		+F	NaN
NaN			NaN	NaN		NaN	NaN
+F	+F		0	0		+F	NaN

6.3.7 SIN

Description: Computes the sine of src0 (in radians) and stores computed result in dst0. Src0 and dst0 are 32-bit FP values.

Format: SIN <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
   int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
   if (EMask.channel[n] == 1) {
            dst.channel[n] = Sin(src0.channel[srcCh])
   }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode	▨	+0	-0	▨	-1 to 1	NaN

6.3.8 COS

Description: Computes the cosine of src0 (in radians) and stores computed result in dst0. Src0 and dst0 are 32-bit FP values.

Format: SIN <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
            dst.channel[n] = Cos(src0.channel[srcCh])
    }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi
Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of COS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode	▨	+1	+1	▨	-1 to 1	NaN

6.3.9 SINCOS

Description: Computes the sine of src0 (in radians) and stores computed result in dst0. Computes the cosine of src0 (in radians) and returns the result to dst1. Src0, dst0 and dst1 are 32-bit FP values.

Format: SINCOS <dst0> <dst1> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
  int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
  if (EMask.channel[n] == 1) {
      if(dst0 != NULL){
              dst0.channel[n] = Sin(src0.channel[srcCh])
          }
          if(dst1 != NULL){
              dst1.channel[n] = Cos(src0.channel[srcCh])
          }
  }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi.

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SINCOS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Notes: See individual Sin and Cos tables for error handling

6.3.10 INT DIV

Description: Computes src0 divided by src1 and returns an integer result to dst0. Src0, src1 and dst0 are 32-bit integers.

Format: INTDIV <dst0> <dst1> <src0> <src1>

Pseudocode:

```
for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
        if(dst0 != NULL){
            dst0.channel[n] = quotient (src0.channel[srcCh] / src1.channel[srcCh])
        }
        if(dst1 != NULL){
            dst1.channel[n] = remainder (src0.channel[srcCh] / src1.channel[srcCh])
        }
    }
}
```

Precision: 32-bit integer

For signed inputs, INT DIV behavior is illustrated by the table below:

Inputs:	**Numerator**	+	+	-	-
	Denominator	+	-	+	-
Outputs:	**Quotient**	+	-	-	+
	Remainder	+	+	-	-

IDIV	**SRC0**		
SRC1	**+ INT**	**- INT**	**0**
+ INT	+INT	-INT	0
- INT	-INT	+INT	0
0	Q:0x7FFF FFFF	Q: 0x8000 0000	Q:0x7FFF FFFF
	R:0x7FFF FFFF	R: 0x8000 0000	R:0x7FFF FFFF
UDIV	**SRC0**		
SRC1	**< > 0**	**0**	
< > 0	UINT	0	
0	Q: 0xFFFF FFFF	Q: 0xFFFF FFFF	
	R: 0xFFFF FFFF	R: 0xFFFF FFFF	

§§

7 Message Gateway

The Message Gateway shared function provides a mechanism for active thread-to-thread communication. Such thread-to-thread communication is based on direct register access. One thread, a **requester thread**, is capable of writing into the GRF register space of another thread, a **recipient thread**. Such direct register access between two threads in a multi-processor environment some time is referred to as **remote register access**. Remote register access may include read or write. GEN4 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 behave of the requester. The requester thread and the recipient thread may be on the same EU or on different EUs.

7.1 Messages

Message Gateway supports such thread-to-thread communication with the following three 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 will block any future remote register writes to this thread.
- **ForwardMsg:** forwards a formatted message (remote register write) from a requester thread to a recipient thread.

7.1.1 Message Descriptor

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

Bit	Description
19	This bit is not part of the shared function specific message descriptor)
18:17	Ignored**:** these bits are not part of the shared function specific message descriptor)
16:15	**Notify.** Send Notification Signal. When the low bit of this field is set, the recipient thread's notification counter is incremented. The high bit is not part of the shared function specific message descriptor. This field is only valid for a ForwardMsg message. It is ignored for other messages.
14	**AckReq.** Acknowledgment Required. When this bit is set, an acknowledgment return message is required. Message Gateway will send a writeback message containing the error code to the requester thread using the post destination register address. When this bit is not set, no writeback message is sent to the requesting thread by Message Gateway, even if an error occurs. This field is valid for OpenGateway, CloseGateway, and ForwardMsg messages. When this bit is set, post destination register must be valid and the response length must be 1. When this bit is not set, 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.
13:2	Reserved: MBZ
1:0	**SubFuncID.** Identify the supported sub-functions by Message Gateway. Encodings are: 00 = **OpenGateway.** Open the gateway for the requester thread. 01 = **CloseGateway.** Close the gateway for the requester thread. 10 = **ForwardMsg.** Forward the formatted message to the recipient thread with the given offset from the recipient's register base. 11 = Reserved.

7.1.2 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 will ignore this message; instead, it will close 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.

7.1.2.1 Message Payload

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:29	Reserved: MBZ
	28:21	**RegBase:** The register base address to be stored in the Message Gateway. It is used to compute the destination GRF register address from the offset field in ForwardMsg. RegBase contains 256-bit GRF aligned register address. Note 1: This field aligns with bits [28:21] of the Offset field of the message payload for ForwardMsg. Note 2: the most significant bit of this field must be zero. Format = U8 Range = [0,127]
	20:11	Reserved: MBZ
	10:8	**Gateway Size:** The range limit for messages through the Message Gateway. The maximal allowed Gateway Size is 32 GRF registers. 000 = 1 GRF Register 001 = 2 GRF Registers 010 = 4 GRF Registers 011 = 8 GRF Registers 100 = 16 GRF Registers 101 = 32 GRF Registers 110 = Reserved 111 = Reserved
	7:0	**Dispatch ID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4	31:16	Reserved: MBZ
	15:0	**Key:** The key to be stored in the thread's entry at the Message Gateway.
M0.3:0		Ignored

7.1.2.2 Writeback Message

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

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000 = *Successful*. No Error (Normal) 001 = *Gateway Size Exceeded*. Attempt to open a gateway with a Gateway Size that is larger than 32 GRF registers 101 = *Opcode Error*. Attempt to send a message which is not either open/close/forward other codes: Reserved

7.1.3 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 completion of the CloseGateway function. Only the least significant DWord in the post destination register is overwritten.

7.1.3.1 Message Payload

DWord	Bit	Description
M0.7:6		Ignored
M0.5	31:8	Ignored
	7:0	**Dispatch ID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.

DWord	Bit	Description
M0.4:0		Ignored

7.1.3.2 Writeback Message

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

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000 = *Successful*. No Error (Normal) 101 = *Opcode Error*. Attempt to send a message which is not either open/close/forward other codes: Reserved

7.1.4 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.

7.1.4.1 Message Payload

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:29	Reserved: MBZ
	28:16	**Offset:** It provides the destination register position in the recipient thread GRF register space as the offset from the RegBase stored in the recipient thread's gateway entry. The offset is in unit of byte, such that bits [28:21] is the 256-bit aligned register offset and bits [4:0] is the sub-register offset. The sub-register offset must be aligned to the Length field in bits [10:8]. The subfields of Offset are further illustrated as the following. Offset[28:21]: Register offset from the gateway base (Range [0, 127]: bit 12 MBZ) Offset[20:18]: DW offset Offset[17:16]: Byte offset (must be 00 for all DW length cases)
	15:11	Reserved: MBZ
	10:8	**Length:** The length of the data segment. 000 = 1 byte 001 = Reserved 010 = 1 dword 011 = 2 dwords 100 = 4 dwords 101-111: Reserved
	7:0	**Dispatch ID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4	31:28	Ignored
	27:24	**EUID:** The Execution Unit ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.
	23:18	Ignored
	17:16	**TID:** The Thread ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.
	15:0	**Key** The key to match with the one stored in the recipient thread's entry in Message Gateway.
M0.3	31:0	**Data Segment DWord 3:** valid only for the 4-DWord data segment length
M0.2	31:0	**Data Segment DWord 2:** valid only for the 4-DWord data segment length
M0.1	31:0	**Data Segment Dword 1:** valid only for the 2- and 4-DWord data segment lengths

DWord	Bit	Description	
M0.0	31:24	**Data Segment Byte 0:** the same byte must be copied to all four positions within this DWord. Valid only for the 1-Byte data segment length.	**Data Segment Dword 0:** valid only for the 1-, 2- and 4-Dword data segment lengths
	23:16	**Data Segment Byte 0**	
	15:8	**Data Segment Byte 0**	
	7:0	**Data Segment Byte 0**	

7.1.4.2 Writeback Message to Requester Thread

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

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000 = *Successful*. No Error (Normal) 001 = Reserved 010 = *Gateway Closed*. Attempt to send a message through a closed gateway 011 = *Key Mismatched*. Attempt to send a message with a mismatching key 100 = *Limit Exceeded*. Attempt to send a message with offset beyond the gateway limit 101 = *Opcode Error.* Attempt to send a message which is not either open/close/forward 110 = *Invalid Message Size.* Attempt to forward a message with length greater than 4 DW 111 = Reserved

7.1.4.3 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.

7.1.5 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 *vol1 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.

7.1.5.1 Message Payload

DWord	Bit	Description
M0.7	31:0	Debug
M0.6	31:0	Debug
M0.5	31	**Return to High GRF:** 0: the return 128-bit data goes to the first half of the destination GRF register 1: the return 128-bit data goes to the second half of the destination GRF register
	30:8	Reserved : MBZ
	7:0	**Dispatch ID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

7.1.5.2 Writeback Message to Requester Thread

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

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

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

DWord	Bit	Description
W0.7:4		Reserved (not overwritten)
W0.3	31:0	**RelativeTimeLapHigh:** This field returns the MSBs of time lap for the relative clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
W0.2	31:20	**RelativeTimeLapLow:** This field returns the LSBs of time lap for the relative clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ
W0.1	31:0	**AbsoluteTimeLapHigh:** This field returns the MSBs of time lap for the absolute clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
W0.0	31:20	**AbsoluteTimeLapLow:** This field returns the LSBs of time lap for the absolute clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ

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

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

§§

8 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.

8.1 URB Size

The URB provides 16KB of storage, arranged as 512 256-bit *rows*. A row corresponds in size to an EU GRF register. Read/write access to the URB is generally supported on a row-granular basis.

A URB *entry* is a logical entity within the URB, referenced by an entry *handle* and comprised of some number of consecutive rows.

8.2 URB Access

The URB can be <u>written</u> 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 <u>read</u> 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.

Note that neither the CPU nor EU threads can read the URB directly.

8.3 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.

8.4 Messages

There is only one type of message supported by the URB shared function: URB_WRITE. It is primarily used by a thread to write data in to an entry in the URB, as referenced by the passed handle.   FF units of the 3D pipeline snoop these messages, and a side effect of the message may be some information being passed to the FF unit which spawned the thread.

This section documents the global aspects of the URB write messages. The actual data contained in the message differs for each fixed function – refer to *3D Pipeline* and the fixed-function chapters or details on 3D URB data formats, *Media* for media-specific URB data formats, and *Graphics Processing Engine* for details on Constant URB Entries (CURBEs).

Programming Notes:

- The **End of Thread** bit in the message descriptor may be set on URB messages only in threads dispatched by the vertex shader (VS), geometry shader (GS), clipper, and strips and fans (SF) units.

8.4.1 Execution Mask

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

8.4.2 Message Descriptor

Bit	Description
19	This bit is not part of the shared function specific message descriptor)
18:16	Ignored
15	**Complete** If clear, this signals that the URB entry(s) referenced by the handle(s) are not yet completely specified. This setting is used to perform partial writes to URB entries, as would be required when writing an entry larger than the maximum single message payload can accommodate. Only the final write would be marked "complete". Partial writes may be unordered. If set, this signals that there will be no further writes (past this one) to the specific URB entry(s) by the thread. A snooping FF unit uses this to identify when the corresponding URB entry(s) are completely specified, at which point the FF unit can initiate further operations the entry(s) (either a readback, passing the handle(s) down the pipeline, or immediate deallocation if the entry is "unused"). This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. **Programming Notes:** The following message descriptor fields are only valid when **Complete** is set: **Used** The following message header fields are only valid when Complete is set: **Handle 0 PrimType, Handle 0 PrimStart, Handle 0 PrimEnd.**
14	**Used** If set, this signals that the URB entry(s) referenced by the handle(s) are valid outputs of the thread. In all likelihood this means that that entry(s) contains complete & valid data to be subject to further processing by the pipeline. If clear, this signals that the URB entry(s) referenced by the handle(s) are not valid outputs of the thread. Use of this setting will result in the handle(s) being immediately dereferenced by the owning FF unit. This setting is to be used by GS or CLIP threads to dereference handles it obtained (either in the initial thread payload or subsequent allocation writebacks) but subsequently determined were not required (e.g., the object was completely clipped out). **Programming Notes:** • Only GS and CLIP threads are permitted to utilize Used==0. All other threads are required (by design) to generate valid outputs in all cases. • This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. • This bit is only valid when **Complete** is set, i.e., it is ignored on partial writes.

Bit	Description
13	**Allocate** If set, this requests that an additional destination URB entry be allocated to the thread by the spawning FF unit. The FF unit will return the handle to this URB entry via a message writeback operation in response to this message (see writeback format below). Therefore, threads must specify a writeback register in 'send' instructions issuing messages with this bit set. If clear, an additional allocation is not requested. **Programming Notes:** • This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. • This bit is valid on all URB_WRITE messages, e.g., it could be used to allocate a new handle on a partial write (**Complete** not set). • Only one Allocate request (per thread) can be outstanding. Upon requesting an allocation, the thread must wait for the handle to be returned (written back) before another allocation can be requested.
12	**Fast Composite Restriction Check Pass** Ignored
11:10	**Swizzle Control.** This field is used to specify which "swizzle" operation is to be performed on the write data. It indirectly specifies whether one or two handles are valid. 00 = URB_NOSWIZZLE The message data is to be written directly to a single URB entry (Handle 0). 01 = URB_INTERLEAVED The message contains data to be written to two URB entries. The message data provided is interleaved such that the upper DWords (7:4) of each 256-bit unit contain data to be written to Handle 1, and the lower DWords (3:0) contain data to be written to Handle 0. The URB shared function will de-interleave this data and write the two separate data streams to the two entries using the single Offset value (see Offset below for more details). 10 = URB_TRANSPOSE This message contains data that is to be "transposed" before being written to the URB. The transpose applied is tailored to the passing of data between the SF and WM stages – it is not a generic transpose. (See description below). Therefore, the assumption is that this mode will only be used by Setup threads, where the setup-result data is swizzled before being written to the URB in order to provide a more optimal format for use in a subsequent PS thread. (See Strip/Fan, Windower chapters). See Programming Restrictions in the URB_TRANSPOSE subsection below. 11 = Reserved
9:4	**Offset.** This field specifies a destination offset (in 256-bit units) from the start of the URB entry(s), as referenced by **URB Return Handle *n***, at which the data (if any) will be written. When URB_INTERLEAVED is used, this field provides a 256-bit granular offset applied to both URB entry destinations. When URB_TRANSPOSE is used, this field provides a 256-bit granular offset applied to the URB entry destination. The least significant bit of **Offset** must be zero.
3:0	**URB Opcode** 0 = URB_WRITE all other codes are Reserved

The following table lists the valid and invalid combinations of the Complete, Used, Allocate and EOT bits:

Complete	Used	Allocate	EOT	Valid?	Usage
0	d/c	0	0	Valid.	Normal partial-write or non-write of URB.
0	d/c	0	1	Valid only if any and all preceding URB entries have been marked as "complete" and there is no outstanding Allocate request.	Thread terminate w/ non-write of URB
0	d/c	1	0	Valid only if any and all preceding URB entries have been marked as "complete" and there is no outstanding Allocate request.	Non-write of URB with request for an additional handle.
0/1	d/c	1	1	Invalid. Thread must never terminate with an outstanding writeback request.	n/a
1	0	0/1	0	Valid	Dereference of URB entry without/with new allocation request.
1	0	0	1	Valid	Dereference of URB entry and thread termination.
1	1	0/1	0	Valid	Completion of URB entry output without/with new allocation request.
1	1	0	1	Valid	Completion of URB entry output and thread termination.

8.4.3 URB_WRITE

8.4.3.1 URB_WRITE Message Header

DWord	Bit	Description
M0.7	31:0	**Debug**
M0.6	31:0	**Debug**
M0.5	31:8	Ignored
	7:0	**FFTID.** The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:26	Ignored
	25:16	Ignored (SO_NUM_PRIMS_WRITTEN is incremented via SVBWrite messages to the DataPort).
	15:7	Ignored
	6:2	**Handle 0 PrimType.** This field associates a primitive type with the vertex written at Handle 0. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
	1	**Handle 0 PrimStart.** This field is used to indicate that the vertex written at Handle 0 is the <u>first</u> vertex of a primitive. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
	0	**Handle 0 PrimEnd.** This field is used to indicate that the vertex written at Handle 0 is the <u>last</u> vertex of a primitive. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
M0.1	31:16	**Handle ID 1.** This ID is assigned by the fixed function unit and links the work in channel 1 to a specific entry within the fixed function unit. This field is ignored unless **Swizzle Control** indicates Interleave mode.
	15:0	**URB Return Handle 1.** This is the URB handle where channel 1's results are to be placed. This field is ignored unless **Swizzle Control** indicates interleave mode.
M0.0	31:16	**Handle ID 0.** This ID is assigned by the fixed function unit and links the work in channel 0 to a specific entry within the fixed function unit.
	15:0	**URB Return Handle 0.** This is the URB handle where channel 0's results are to be placed.

8.4.3.2 URB_WRITE Message Payload

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

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

Payload	Usage
URB_NOSWIZZLE	The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The message payload contains data to be written to two separate URB entries. The payload data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.
URB_TRANSPOSE	The message payload contains data that is to be transposed before being written to the URB. See the *Strip & Fan (SF) Unit* chapter for details on the source and destination data layouts and intended usage model.

8.4.3.2.1 URB_NOSWIZZLE

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

Programming Notes:

- The URB function will ignore the Channel Enables associated with this message and write all channels into the URB.

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

DWord	Bit	Description
M1.7	31:0	**Vertex Data [7]**
M1.6	31:0	**Vertex Data [6]**
M1.5	31:0	**Vertex Data [5]**
M1.4	31:0	**Vertex Data [4]**
M1.3	31:0	**Vertex Data [3]**
M1.2	31:0	**Vertex Data [2]**
M1.1	31:0	**Vertex Data [1]**
M1.0	31:0	**Vertex Data [0]**
M2.7	31:0	**Vertex Data [15]**
M2.6	31:0	**Vertex Data [14]**
M2.5	31:0	**Vertex Data [13]**
M2.4	31:0	**Vertex Data [12]**
M2.3	31:0	**Vertex Data [11]**
M2.2	31:0	**Vertex Data [10]**
M2.1	31:0	**Vertex Data [9]**
M2.0	31:0	**Vertex Data [8]**
…		**…**
Mn.7	31:0	**Vertex Data [8(n-2)+7]**
Mn.6	31:0	**Vertex Data [8(n-2)+6]**
Mn.5	31:0	**Vertex Data [8(n-2)+5]**
Mn.4	31:0	**Vertex Data [8(n-2)+4]**
Mn.3	31:0	**Vertex Data [8(n-2)+3]**
Mn.2	31:0	**Vertex Data [8(n-2)+2]**
Mn.1	31:0	**Vertex Data [8(n-2)+1]**
Mn.0	31:0	**Vertex Data [8(n-2)+0]**

8.4.3.2.2 URB_INTERLEAVED

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

Programming Restrictions:

- At least 256 bits per vertex (512 bits total, M1 & M2) must be written. Writing only 128 bits per vertex (256 bits total, M1 only) results in UNDEFINED operation.
- The URB function will use (not ignore) the Channel Enables associated with this message.

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

8.4.3.2.3 URB_TRANSPOSE

The following table shows an example layout of a URB_TRANSPOSE payload and how the data is transposed and stored in the destination URB entry. Note that Source Row 0, Source Row 1, and implied row of all-zero, and Source Row 3 is transposed and stored in successive 4-DW locations in the destination. This is then repeated for the next 3 rows of the source payload. For the intended usage model in the Setup thread, Source Row 0 would contain "Cx" coefficients for the first 8 attributes, Source Row 1 would contain "Cy" coefficients for the first 8 attributes, and Source Row 2 would contain "C0" coefficients for the first 8 attributes, then repeating for the next 8 attributes. Insertion of the implied all-zero row is required to align the Cx,Cy and C0 attributes into half-rows within the URB. This permits the used of the "LINE" instruction to initiate attribute interpolation in the subsequent PS thread.

Programming Notes:

- The message payload must contain a multiple of 3 Source Rows of data (excluding the message header).
- The URB function will ignore the Channel Enables associated with this message and write all channels into the URB.

Table 8-1. URB_TRANSPOSE Payload

DWord	Bit	Description
M1.0-7	31:0	**Source Row 0 (e.g., Cx coeffs for the 1st set of 8 attributes)**
M2.0-7	31:0	**Source Row 1 (e.g., Cy coeffs for the 1st set of 8 attributes)**
M3.0-7	31:0	**Source Row 2 (e.g., C0 coeffs for the 1st set of 8 attributes)**
M4.0-7	31:0	**Source Row 3 (e.g., Cx coeffs for the 2nd set of 8 attributes)**
M5.0-7	31:0	**Source Row 4 (e.g., Cy coeffs for the 2nd set of 8 attributes)**
M6.0-7	31:0	**Source Row 5 (e.g., C0 coeffs for the 2nd set of 8 attributes)**
…	31:0	**…**

Table 8-2.URB_TRANSPOSE URB Destination Layout

	URB DW							
URB Row	7	6	5	4	3	2	1	0
n+0	M3.1	0	M2.1	M1.1	M3.0	0	M2.0	M1.0
n+1	M3.3	0	M2.3	M1.3	M3.2	0	M2.2	M1.2
n+2	M3.5	0	M2.5	M1.5	M3.4	0	M2.4	M1.4
n+3	M3.7	0	M2.7	M1.7	M3.6	0	M2.6	M1.6
n+4	M6.1	0	M5.1	M4.1	M6.0	0	M5.0	M4.0
n+5	M6.3	0	M5.3	M4.3	M6.2	0	M5.2	M4.2
n+6	M6.5	0	M5.5	M4.5	M6.4	0	M5.4	M4.4
n+7	M6.7	0	M5.7	M4.7	M6.6	0	M5.6	M4.6
	…	…	…	…	…	…	…	…

8.4.3.3 Writeback Message for URB Entry Allocate

A writeback only occurs if the **Allocate** bit is set in the message descriptor. A single register is returned containing the URB Return Handle and Handle ID for the allocated handle in the low DWord is returned. All high DWords contain zero.

DWord	Bit	Description
W0.7:1		Reserved : MBZ
W0.0	31:16	**Handle ID.** This ID is assigned by the fixed function unit and links the thread to a specific entry within the fixed function unit.

§§

9 Execution Unit ISA

9.1 Introduction

9.1.1 Objective and Scope

The core of GEN4 architecture consists of an array of multi-threaded processors, also referred to as Execution Units (EU). This Instruction Set Architecture (ISA) document specifies the instructions executable on the EUs of the GEN4 architecture. It defines the data types in the GEN4 architecture. It includes the binary format (machine code) and ASCII format (native syntax) of each instruction. It also provides example usages of instructions and modes of instructions, and certain data formats. The programming guideline in appendix provides information to help developers to understand the usage of GEN4 ISA. However, it is not intended to be a comprehensive tutorial.

9.1.2 Terms and Acronyms

AIP	Application IP. This is part of the control registers for exception handling for a thread. Upon an exception, hardware moves the current IP into this register and then jumps to SIP.
ARF	Architecture Register File. It is a collection of architecturally visible registers for a thread such as address registers, accumulator, flags, notification registers, IP, null, etc. ARF should not be mistaken as just the address registers.
B	Byte. As a numerical data type of 8 bits, B represents a signed byte integer. It is used to specify the type of an operand in an instruction.
BNF	Backus Naur Form, a formal notation to describe the syntax of a given language. The meta symbols of BNF include "**::=**", "**|**", and "**< >**", where "::=" means "is defined as"; "|" means "or"; and angle brackets "<" and ">" are used to surround category names.
CR	Control Register. These read-write registers are used for thread mode control and exception handling for a thread.
D	Double word (DWord). As a fundamental data type, D or DW represents 4 bytes. It may be used to specify the type of an operand in an instruction.

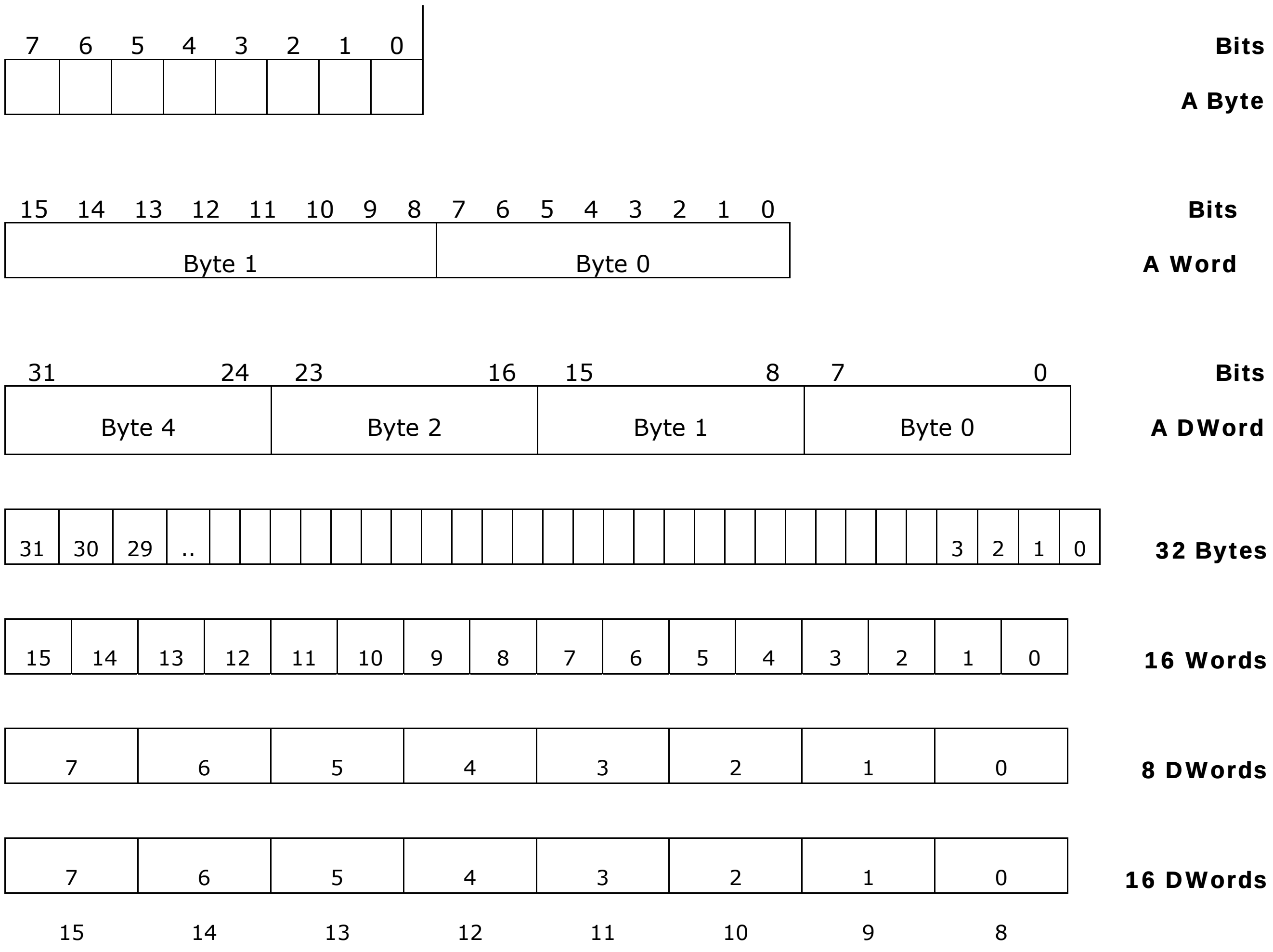
EOT	End Of Thread. This is a message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set in order to properly terminate.
EU	Execution Unit. An EU is a multi-threaded processor within the GEN4 multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a GEN4 Core.
EUID	Execution Unit Identifier. The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU where a thread is located. A thread can be uniquely identified by the EUID and TID.
ExecSize	Execution Size.
Execution Size	Execution Size indicates the number of data elements processed by a GEN4 SIMD instruction. It is one GEN4 instruction field and can be changed at a per instruction level.
FLT_MAX	The magnitude of the maximum represent-able single-precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's.
fmax	Same as FLT_MAX.
GEN4 Core	Alternative name for an EU in the GEN4 multi-processor system.
GRF	General Register File. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.
ISA	Instruction Set Architecture. The GEN4 ISA describes the instructions supported by a GEN4 EU. A sequence of GEN4 instructions forms a thread executed on an EU.
JIT	Just-In-Time compiler
LSB	Least Significant Bit
Message	Messages are data packages transmitted from a thread to another thread, to another shared function or to another fixed function. Message passing is the primary communication mechanism of the GEN4 architecture.
MRF	Message Register File. This is the write-only register space, organized as an array of 256-bit registers, for a thread to communicate with shared functions or other threads.
MSB	Most Significant Bit

DQ	Double Quad word (DQword). As a fundamental data type, DQ represents 16 bytes.
POR	Plan Of Record
QW	Quad Word (QWord). As a fundamental data type, QW represents 8 bytes.
QQ	Quad Quad word (QQword). As a fundamental data type, QQ represents 32 bytes.
Sub-Register	Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, *r2*, is a 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*.
SIMD	Single Instruction Multiple Data. The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at the instruction level. It can also be used to describe the instructions in such an architecture.
SIP	System IP. There is one global System IP register for all the threads. From a thread's point of view, this is a virtual read-only register. Upon an exception, hardware performs certain book-keeping functions and then jumps to SIP.
SR	State Register. The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP.
Thread	A thread is an instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction – a send instruction with EOT that signals the thread termination. Threads in the GEN4 system may be independent from each other or communicate with each other through the Message Gateway share function.
TID	Thread Identifier. The 2-bit field within a thread state register (SR0) that identifies which out of the four possible thread slots on the EU is executing that thread. A thread can be uniquely identified by the EUID and TID.
TS	Thread Spawner. TS is the second and the last fixed function stage of the media pipeline.
V	Immediate integer vector. As a numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4 bits each. The 4-bit integer element is in 2's complement form. It may be used to specify the type of an immediate operand in an instruction.

VF	Immediate floating point vector. As a numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.
W	Word. As a numerical data type of 16 bits, W represents a signed word integer. It is used to specify the type of an operand in an instruction.
URB	Unified Return Buffer. The on-chip memory managed/shared by GEN4 Fixed Functions. Threads use the URB to return data that will be consumed either by a Fixed Function or other threads.
UB	Unsigned Byte integer. A numerical data type of 8 bits. It may be used to specify the type of an operand in an instruction.
UD	Unsigned Double Word integer. A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.
UW	Unsigned Word integer. A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.
VFE	Video Front End. VFE is the first fixed function stage of the media pipeline.

9.1.3 Formats and Conventions

In order to conveniently (and without ambiguity) describe the register files with 256-bit wide registers that may contain various data types with different data element widths, it is important to use a consistent table format to represent the registers. Throughout this document, we will adopt the following table formats and conventions. When a register or a number is presented by a row, increasing order is always **from right to left** and then **top down** pictorially. In other words, for a bit field, the LSB to MSB is from right to left; for a byte sequence, the least significant byte to the most significant byte is also from right to left. This is consistent with the 'Little Endian' convention used by IA-32 machines. The following tables depict the layout formats for different data units.



With this convention, we note that the execution channels are logically viewed as from right to left too, which is a little bit unconventional. However, as shown in the *GEN4 Execution Environment* Chapter, it matches with the bit order of the flag registers. This also impacts the view of a GRF register region, now the region origin is located at the upper-right corner and a region row is viewed from right to left.

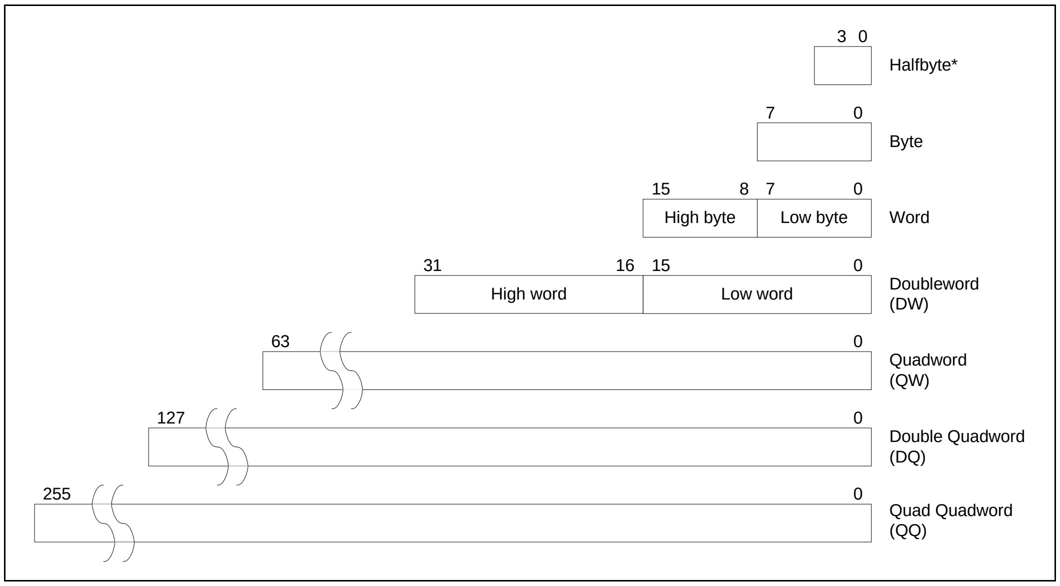
§§

10 EU Data Types

10.1 Fundamental Data Types

The fundamental data types in the GEN4 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 Figure 10-1, a halfbyte contains 4 bits, a byte contains 8 bits, a word contains two bytes, and a doubleword (dword) contains two words, and so on. Halfbyte is a special data type such that it cannot be accessed directly as standalone data element. It is only allowed as a subfield of the numerical data type of "packed signed halfbyte integer vector" described in the next section.

Figure 10-1. Fundamental data types



With the exception of halfbyte, the access of a data element to/from a GEN4 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 unit of byte. 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. Quadword, double quadword and quad quadword do not have corresponding numerical data type. Instead, they are used to describe a group (a vector) of numerical data elements of smaller size align to larger natural boundaries.

10.2 Numerical Data Types

The numerical data types defined in the GEN4 architecture include signed and unsigned integers and floating-point numbers (floats) of various numbers of bits. These numerical data types are pictorially illustrated in Figure 10-2 and Figure 10-3. Table 10-1 details the notation, size and numerical range of each data type. The largest numerical data type has 32 bits.

Figure 10-2. Integer numerical data types

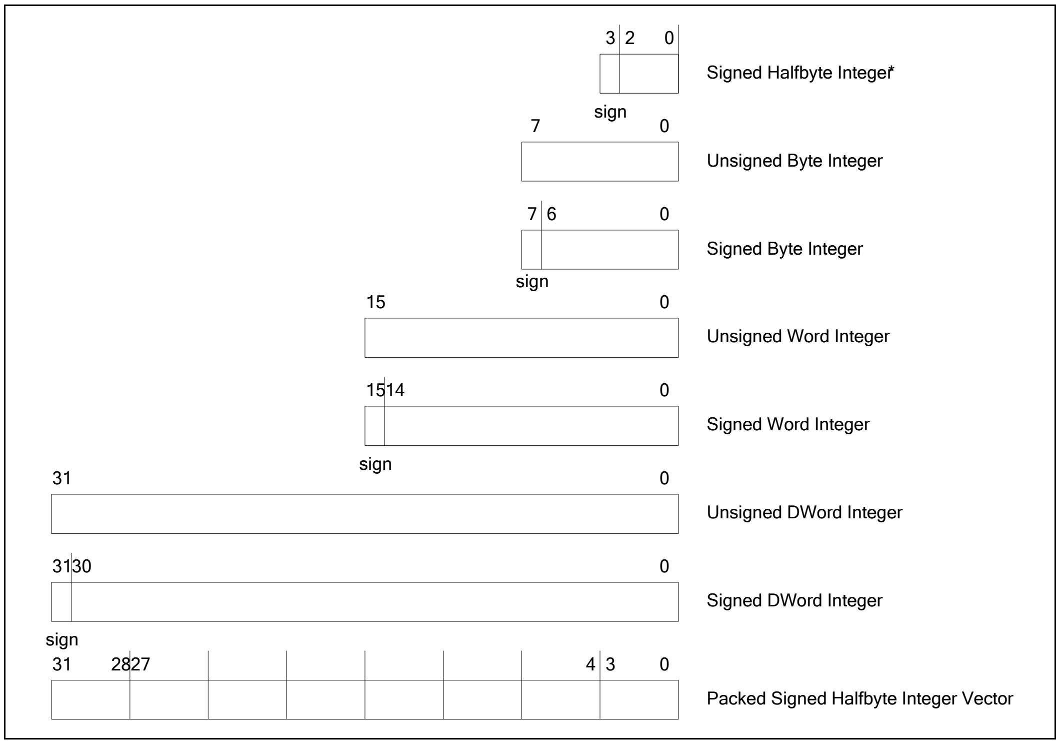


Figure 10-3. Floating point numerical data types

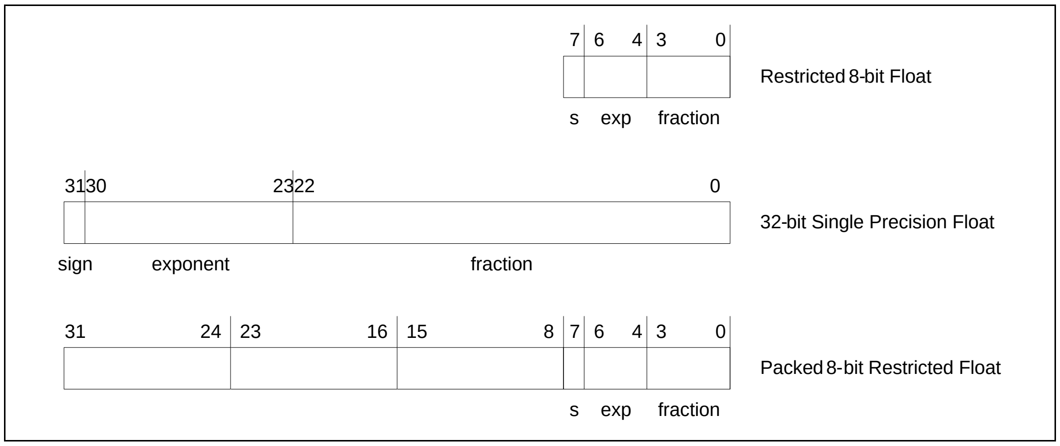


Table 10-1. Formats and ranges of numerical data types

Notation	Numerical Data Types	Fundamental Data Type	Range
UB	Unsigned Byte Integer	Byte	[0, 255]
B	Signed Byte Integer	Byte	[-128, 127]
UW	Unsigned Word Integer	Word	[0, 65535]
W	Signed Word Integer	Word	[-32768, 32767]
UD	Unsigned Doubleword Integer	Doubleword	[0, $2^{32} - 1$]
D	Signed Doubleword Integer	Doubleword	[-2^{31}, $2^{31} - 1$]
F	Single Precision Float	Doubleword	[$-(2-2^{-23})^{127}$…-2^{-149}, 0.0, 2^{-149}… $(2-2^{-23})^{127}$]
n/a	Signed Halfbyte Integer	Halfbyte	[−8, 7]
V	Packed Signed Halfbyte Integer Vector	Doubleword	[−8, 7]
n/a	Restricted 8-bit Float	Byte	[−31…−0.125, 0, 0.125… 31]
VF	Packed Restricted Float Vector	Doubleword	[−31…−0.125, 0, 0.125… 31]

10.2.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, a word or a doubleword. The range for an unsigned byte integer is from 0 to 255. The range for an unsigned word integer is from 0 to 65535. The range for an unsigned doubleword integer is from 0 to $2^{32} - 1$.

The short hand notation for an unsigned byte integer, an unsigned word integer, and an unsigned doubleword integer is **UB**, **UW**, **UD**, respectively.

10.2.2 Signed Integers

Signed integers are signed binary number in 2's complement form contained in a halfbyte, a byte, a word or a doubleword. A signed halfbyte integer has a numerical range from −8 to 7 with the sign bit at bit 3. A signed byte integer has a range from −128 to 127 with the sign at bit 7. A signed word integer is has a range from -32768 to 32767 with the sign at bit 15. A signed doubleword integer has a range from -2^{31} to $2^{31} - 1$ with the sign at bit 31.

The short hand notation for a signed byte integer, a signed word integer, and a signed doubleword integer is **B**, **W**, **D**, respectively.

10.2.3 Single Precision Floating-Point Numbers

The single precision floating point numbers is contained in a doubleword. Floating point format is as defined in IEEE Standard 754 for Binary Floating-Point Arithmetic. Maximal representable number is $(2–2^{-23})^{127}$ and the minimal number is $– (2–2^{-23})^{127}$. The smallest fractional negative number $–2^{-149}$ and the smallest fractional positive number is 2^{-149}. Value 0.0 has no fractional parts.

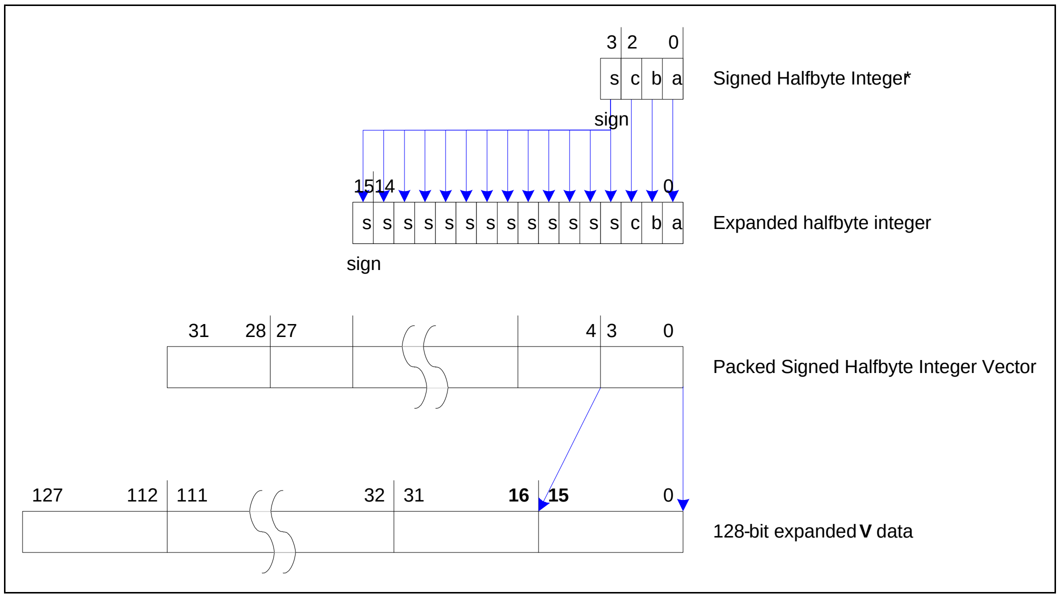
The short hand format notation for a single precision floating-point number is **F**.

10.2.4 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 numerical data type is only used by an immediate source operand of doubleword in a GEN4 instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN4 hardware converts the 32-bit vector into 8-element signed word vector by sign extension. This is illustrated in Figure 10-4.

The short hand format notation for a packed signed half-byte vector is **V**.

Figure 10-4. Converting a Packed Half-byte Vector to a 128-bit Signed Integer Vector



10.2.5 Packed 8-bit 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 numerical data type is only used by an immediate source operand of doubleword in a GEN4 instruction. It cannot be used for the destination operand, or a non-immediate source operand.

Figure 10-5 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.

Figure 10-5. Conversion from a Restricted 8-bit Float to a Single-Precision Float

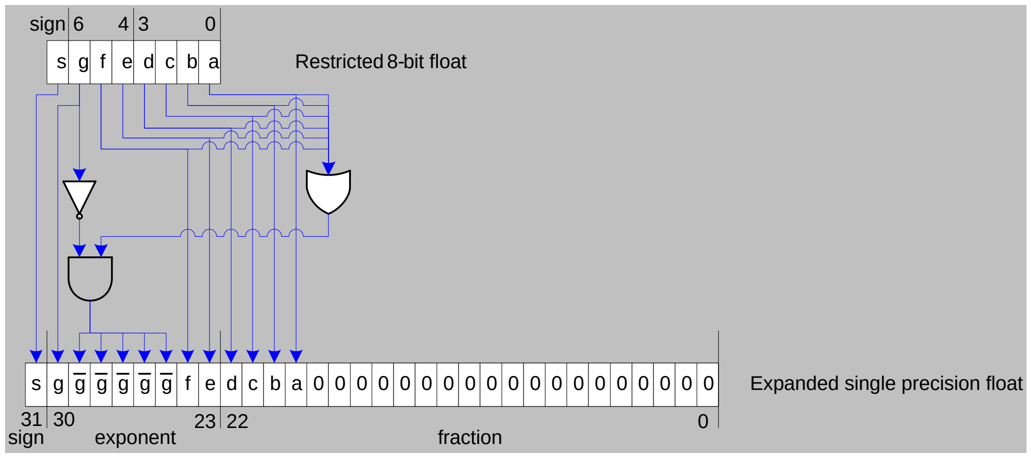

sign
6
4
3
0
s g f e d c b a
Restricted 8-bit float
s g $\overline{g}$ $\overline{g}$ $\overline{g}$ $\overline{g}$ $\overline{g}$ f e d c b a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Expanded single precision float
31
30
23
22
0
sign
exponent
fraction

Table 10-2 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.

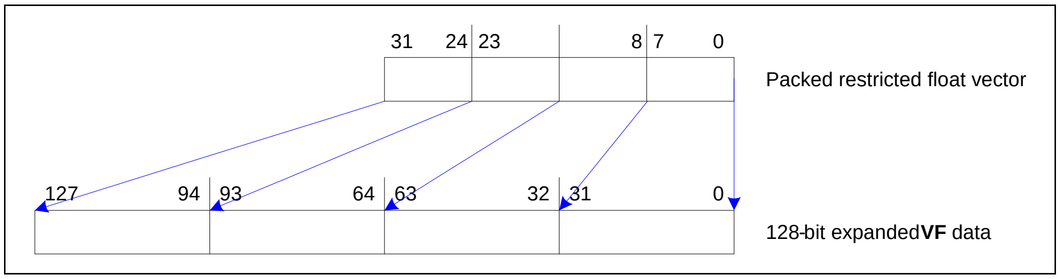
Table 10-2. Example of restricted 8-bit float numbers

Class	Restricted 8-bit Float				Extended 8-bit Exponent	Floating number in decimal
	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]		
Positive Normalized Float	0x70-0x7F	0	**111**	0000 … 1111	**1**000 00**11**	16 … 31
	0x60-0x6F	0	**110**	0000 … 1111	**1**000 00**10**	8 … 15.5
	0x50-0x5F	0	**101**	0000 … 1111	**1**000 00**01**	4 … 7.75
	0x40-0x4F	0	**100**	0000 … 1111	**1**000 00**00**	2 … 3.875
	0x30-0x3F	0	**011**	0000 … 1111	**0**111 11**11**	1 … 1.9375
	0x20-0x2F	0	**010**	0000 … 1111	**0**111 11**10**	0.5 … 0.96875
	0x10-0x1F	0	**001**	0000 … 1111	**0**111 11**01**	0.25 … 0.484375
	0x01-0x0F	0	**000**	0001 … 1111	**0**111 11**00**	0.125 … 0.2421875
	0x00	0	**000**	**0000**	0000 0000	0 (+zero)
Negative Normalized Float	0xF0-0xFF	1	**111**	0000 … 1111	**1**000 00**11**	-16 … -31
	0xE0-0xEF	1	**110**	0000 … 1111	**1**000 00**10**	-8 … -15.5
	0xD0-0xDF	1	**101**	0000 … 1111	**1**000 00**01**	-4 … -7.75
	0xC0-0xCF	1	**100**	0000 … 1111	**1**000 00**00**	-2 … -3.875
	0xB0-0xBF	1	**011**	0000 … 1111	**0**111 11**11**	-1 … -1.9375
	0xA0-0xAF	1	**010**	0000 … 1111	**0**111 11**10**	-0.5 … -0.96875
	0x90-0x9F	1	**001**	0000 … 1111	**0**111 11**01**	-0.25 … -0.484375
	0x81-0x8F	1	**000**	0001 … 1111	**0**111 11**00**	-0.125 … -0.2421875
	0x80	1	**000**	**0000**	0000 0000	-0 (-zero)

Figure 10-6 shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The short hand format notation for a packed signed half-byte vector is **VF**.

Figure 10-6. Converting a Packed Restricted Float Vector to a 128-bit Float Vector



10.3 Floating Point Modes

GEN4 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.

These two modes are supported by all units that perform floating point computations, including GEN4 execution units, GEN4 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.

10.3.1 IEEE Floating Point Mode

10.3.1.1 Partial Listing of Honored IEEE-754 Rules

Here is a summary of expected 32-bit floating point behaviors in GEN4 architecture. Refer to IEEE-754 for topics not mentioned.

- INF – INF = NaN
- 0 * (+/−)INF = NaN
- 1 / (+INF) = +0 and 1 / (−INF) = −0
- (+/−)INF / (+/−)INF = NaN as A/B = A * (1/B)
- INV (+0) = RSQ (+0) = +INF, INV (−0) = RSQ (−0) = −INF, and SQRT (−0) = −0
- RSQ (−finite) = SQRT (−finite) = NaN
- LOG (+0) = LOG (−0) = −INF, LOG (−finite) = LOG (−INF) = NaN
- NaN (any OP) any-value = NaN with one exception for min/max mentioned below. Resulting NaN may have different bit pattern than the source NaN.

- Normal comparison with conditional modifier of EQ, GT, GE, LT, LE, when either or both operands is NaN, returns FALSE. Normal comparison of NE, when either or both operands is NaN, returns TRUE.
  - Note: Normal comparison is either a **cmp** instruction or an instruction with conditional modifier
- Special comparison **cmpn** with conditional modifier of EQ, GT, GE, LT, LE, when the second source operand is NaN, returns TRUE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns FALSE. **Cmpn** of NE, when the second source operand is NaN, returns FALSE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns TRUE.
  - This is used to support the proposed IEEE-754R rule on **min** or **max** operations. For which, if only one operand is NaN, min and max operations return the other operand as the result.
- Both normal and special comparisons of any non-NaN value against +/– INF return exact result according to the conditional modifier. This is because that infinities are exact representation in the sense that +INF = +INF and –INF = –INF.
  - NaN is unordered in the sense that NaN != NaN.

10.3.1.2 Complete Listing of Deviations or Additional Requirements vs. IEEE-754

For a result that cannot be represented precisely by the floating point format, GEN4 execution unit uses rounding toward zero (which is a bit-field truncation of the magnitude portion of a floating point data in sign-magnitude form) to produce a result to the closest representable value. This ends up with a result that is within 1 Unit-Last-Place (1 ULP) of the infinitely precise result.

- GEN4 execution unit can report floating point overflow and NaN into conditional flags. However, there is no support for floating point exceptions, status bits or traps.
- Denorms are flushed to sign-preserved zero on input and output of any floating point mathematical operation.
  - The exception to the above point about flushing denorms is that any I/O or data movement operation that does not manipulate the data (such as point sampling float data, or executing any raw "mov" instruction, or any sort of conditional raw "mov" if present) must not alter data at all (so a denorm remains denorm). Note that doing something that amounts to just moving data, but isn't explicitly, such as multiplying a number by 1.0f is not detected as just a raw "mov", and in this case a denorm flush would happen.
- NaN input to an operation obviously 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/MRF registers.

- API Shader divide operations are implemented as x*(1.0f/y). With the two-step method, x*(1.0f/y), the multiply and the divide each independently operate at the 32-bit floating point precision level (accuracy to 1 ULP).
- See the Type Conversion section below for rules on converting to/from float representations.

10.3.1.3 Comparison of Floating Point Numbers

The following tables (Table 10-3 through Table 10-8) detail the rules for floating point comparison. In the tables, "+/-Fin" stands for a positive or negative finite precision floating point number. Result is either a true (T) or false (FALSE or 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.

Table 10-3. Results of "Greater-Than" Comparison – CMP.G

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+Fin	T	T	T	T	T	T	T/F	FALSE	FALSE
+inf	T	T	T	T	T	T	T	FALSE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-4. Results of "Less-Than" Comparison – CMP.L

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	FALSE
-Fin	FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
-0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-5. Results of "Equal-To" Comparison – CMP.E

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
-0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-6. Results of "Not-Equal-To" Comparison – CMP.NE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	T
-Fin	T	T/F	T	T	T	T	T	T	T
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
-0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin	T	T	T	T	T	T	T/F	T	T
+inf	T	T	T	T	T	T	T	FALSE	T
NaN	T	T	T	T	T	T	T	T	T

Table 10-7. Results of “Less-Than Or Equal-To” Comparison – CMP.LE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	FALSE
-Fin	FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm	FALSE	FALSE	T	T	T	T	T	T	FALSE
-0	FALSE	FALSE	T	T	T	T	T	T	FALSE
+0	FALSE	FALSE	T	T	T	T	T	T	FALSE
+denorm	FALSE	FALSE	T	T	T	T	T	T	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-8. Results of “Greater-Than or Equal-To” Comparison – CMP.GE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	T	T	T	T	T	T	FALSE	FALSE	FALSE
-0	T	T	T	T	T	T	FALSE	FALSE	FALSE
+0	T	T	T	T	T	T	FALSE	FALSE	FALSE
+denorm	T	T	T	T	T	T	FALSE	FALSE	FALSE
+Fin	T	T	T	T	T	T	T/F	FALSE	FALSE
+inf	T	T	T	T	T	T	T	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

10.3.1.4 Min / Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in GEN4 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 one 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 a NaN, which may not have the same binary form as any of the two numbers.

When using CMPN for MIN/MAX, the flag polarity for CMPN and SEL instructions must be the same:

Evaluations	GEN4 Instructions
MIN(src0, src1) = (src0 < src1) ? src0 : src1	cmpn.l.f0.0 null src0 src1 (f0.0) sel dst src0 src1
MAX(src0, src1) = (src0 >= src1) ? src0 : src1	cmpn.ge.f0.0 null src0 src1 (f0.0) sel dst src0 src1

The following tables (Table 10-9 through Table 10-14) 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'.

Table 10-9. Results of "Greater-Than" Comparison-NaN – CMPN.G

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+Fin	T	T	T	T	T	T	T/F	FALSE	T
+inf	T	T	T	T	T	T	T	FALSE	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-10. Results of "Less-Than" Comparison-NaN – CMPN.L

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	T
-Fin	FALSE	T/F	T	T	T	T	T	T	T
-denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
-0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-11. Results of "Equal-To" Comparison-NaN – CMPN.E

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
-0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-12. Results of "Not-Equal-To" Comparison-NaN – CMPN.NE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	**FALSE**
-Fin	T	T/F	T	T	T	T	T	T	**FALSE**
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
-0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+Fin	T	T	T	T	T	T	T/F	T	**FALSE**
+inf	T	T	T	T	T	T	T	FALSE	**FALSE**
NaN	T	T	T	T	T	T	T	T	**FALSE**

Table 10-13. Results of "Less-Than Or Equal-To" Comparison-NaN – CMPN.LE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	T
-Fin	FALSE	T/F	T	T	T	T	T	T	T
-denorm	FALSE	FALSE	T	T	T	T	T	T	T
-0	FALSE	FALSE	T	T	T	T	T	T	T
+0	FALSE	FALSE	T	T	T	T	T	T	T
+denorm	FALSE	FALSE	T	T	T	T	T	T	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-14. Results of "Greater-Than or Equal-To" Comparison-NaN – CMPN.GE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	T	T	T	T	T	T	FALSE	FALSE	T
-0	T	T	T	T	T	T	FALSE	FALSE	T
+0	T	T	T	T	T	T	FALSE	FALSE	T
+denorm	T	T	T	T	T	T	FALSE	FALSE	T
+Fin	T	T	T	T	T	T	T/F	FALSE	T
+inf	T	T	T	T	T	T	T	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

10.3.2 Alternative Floating Point Mode

The key characteristics of the alternative floating point mode is that NaN, Inf and denorm are not expected for an application to pass into the graphics pipeline, and the graphics hardware must not generate NaN, Inf or denorm as computation result. For example, a result that is larger than the maximum representable floating point number is expected to be flushed to the largest representable floating point number, i.e., +FLT_MAX. The FLT_MAX has an exponent of 0xFE and a mantissa of all one's, which is the same for IEEE floating point mode.

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 +/- FLT_MAX, 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.

Table 10-15 shows the support of these differences in various hardware units.

Table 10-15. Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	Sampler	RC
Any +/- INF result flushed to +/- FLT_MAX	Y	Y	Y	Y	Y	Y	Y	Y
Log, rsq, sqrt take abs() of sources	N/A	N/A	N/A	N/A	N/A	Y	N/A	N/A

Table 10-16 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.

Table 10-16. Dismissed legacy behaviors

Suggested IEEE-754 Deviations	Reason for Dismiss
Mov forces (+/-)INF to (+/-)FLT_MAX	(+/-)INF is never present as input
(+/-)INF – (+/-)INF = +/- FLT_MAX instead of NaN	(+/-)INF is never present as input
Denorm must be flushed to zero in all cases (including trivial mov and point sampling)	Denorm is never present as input
Anything*0=0 (including NaN*0=0 and INF*0=0)	NaN and INF are never present as input
Except propagated NaN, NaN is never generated	NaN is never present as input and GEN4 never generates NaN based on rules in the previous table
An input NaN gets propagated excepting (a)-(d)	NaN is never present as input
(a) Rcp (and rsq) of 0 yields FLT_MAX	N/A, as it is already covered by the general rule "Any +/- INF result flushed to +/- FLT_MAX"
(b) Sampler honors 0/0 = 0 as if (1/0)*0	There is no divide in Sampler
I Rcp (and rsq) of INF yields +/- 0	(+/-)INF is never present as input
(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0	There is no divide in Sampler

10.4 Type Conversion

10.4.1 Float to Integer

Converting from float to integer is based on rounding toward zero. If the floating point value is +0, -0, +Denorm, -Denorm, +NaN –r -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UD, D, UW, W, UB and B).

Input Format	Output Format					
F	UD	D	UW	W	UB	B
+/- Zero	00000000	00000000	00000000	00000000	00000000	00000000
+/- Denorm	00000000	00000000	00000000	00000000	00000000	00000000
NAN	00000000	00000000	00000000	00000000	00000000	00000000
-NAN	00000000	00000000	00000000	00000000	00000000	00000000
INF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-INF	00000000	80000000	00000000	00008000	00000000	00000080
$+2^{32}$ (*)	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
$-2^{32}-1$ (*)	00000000	80000000	00000000	00008000	00000000	00000080

10.4.2 Integer to Integer with Same or Higher Precision

Converting an unsigned integer to a signed or an unsigned integer with higher precision is based on zero extension.

Converting an unsigned integer to a signed integer with the same precision is based on modular wrap-around. Without saturation, a larger than represent-able number becomes a negative number. With saturation, a larger than represent-able number is saturated to the largest positive represent-able number.

Converting a signed integer to a signed integer with higher precision is based on sign extension.

Converting a signed integer to an unsigned integer with higher precision is based on zero 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.

10.4.3 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.

10.4.4 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, LSBs are truncated. This truncation is performed in sign-magnitude domain, thus, is equivalent to floating point rounding toward zero operation.

11 Execution Environment

11.1 Overview

GEN4 instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Supports for 3D graphics API (application program interface) Shader instructions are mostly native, meaning that GEN4 provides efficient execution for Shader programs. Depending on the 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 GEN4 native instructions may be required. In addition, there are many specific capabilities to accelerate media applications. The following list provides a summary of the GEN4 instruction set.

- GEN4 ISA support SIMD (single instruction multiple data) instructions. The number of data elements per instruction depends on the data type.
- GEN4 ISA supports SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- GEN4 ISA supports instruction level variable-width SIMD execution.
- GEN4 ISA supports conditional SIMD execution via destination mask, predication, and execution mask.
- GEN4 ISA supports in-place format conversion and mixed data type computations.
- GEN4 ISA supports instruction compression.
- A GEN4 instruction may be executed in multiple cycles over a SIMD execution pipeline.
- Most GEN4 instructions have three operands. Some instructions have additional implied source and destination operands. Some instructions have explicit dual destinations.
- GEN4 ISA supports region-based register addressing.
- GEN4 ISA supports direct and indirect (indexed) register addressing.
- GEN4 instructions may have a scalar and 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).

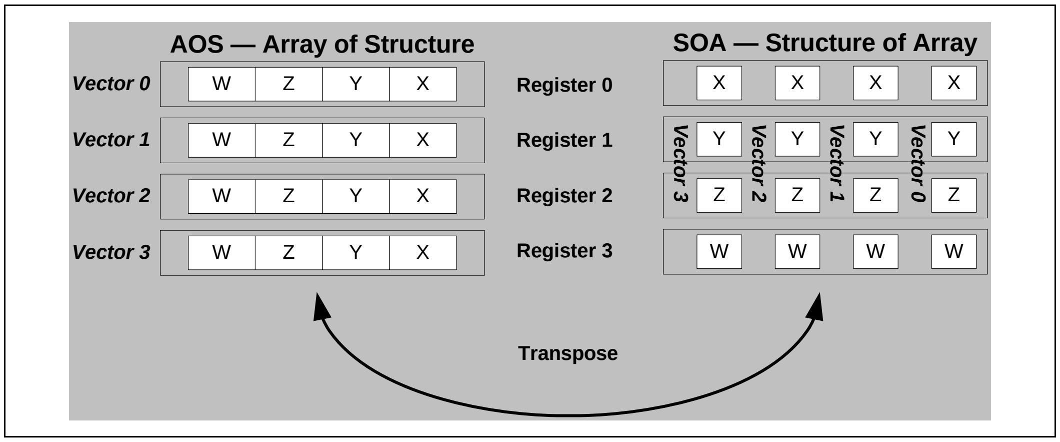
11.2 Primary Usage Models

In describing the usage models of GEN instruction set, it is inevitable to forward reference terminology, syntax and instructions detailed later in this specification. For clarity reasons, not all forward references will be provided in this section as well as subsequent sections. For example, reference to binary instruction fields such as ***Align1***, ***Align16***, ***Compr***, ***SecHalf***, etc., can be found in the Instruction Summary chapter. And assembly instruction syntax can be found in the Instruction Summary chapter and Instruction Reference chapter.

11.2.1 AOS and SOA Data Structures

With the Align1 and Align16 access modes, GEN4 instruction set provides effective SIMD computation regardless whether data are arranged in array of structure (AOS) form or in structure of array (SOA) form. The AOS and SOA data structures are illustrated by the examples in Figure 11-1. The example shows two different ways of storing four vectors in four SIMD registers. For simplicity, data vector and 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. It is obvious in this case the two structures can be related by a matrix transpose operation.

Figure 11-1. AOS and SOA data structures



GEN4 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 structure and run on SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA structure and run on SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA structure, and occasionally in AOS structure with possible mixed mode of operations that use 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 will be used in the following sections to illustrate various modes of operations:

```
add   <dst>.xyz   <src0>.yxzw   <src1>.zwxy
```

This example is an SIMD instruction that takes two source operands <src0> and <src1>, performs addition operation (add), and store the additions 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 swizzle modifier (.yxzw for <src0> and .zwxy for <src1> and destination mask modifier (.xyz). Please refer to programming specifications of 3D graphics API Shader instructions for more details.

A physical GRF register has 256 bits, which may be used to store 8 floating point data elements. For 3D graphics usage, the mode of operation is (loosely) termed after the data structure as SIMD*m*x*n*, where "*m*" is a numerical term describing the size of vector and "*n*" is the number of concurrent program flows executed in SIMD.

- Execution with AOS data structures
  - **SIMD4 (short for SIMD4x1)** stands for the mode of operation where a SIMD instruction operates on 4-element vectors stored packed in the registers. There is only one program flow.
  - **SIMD4x2** standards for the SIMD operation based on a pair of 4-element vectors stored in a register. 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
  - **SIMD8 (short for SIMD1x8)** standards for the SIMD operation based on the SOA data structure where one register contains one data element (the same one) of 8 vectors. Effectively, there are 8 concurrent program flows.
  - **SIMD16 (short for SIMD1x16)** is a special term indicating the use of instruction compression whereas each compressed SIMD instruction operates on a pair of registers that contains one data element (the same one) of 16 vectors. SIMD16 has 16 concurrent program flows.

11.2.2 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 GEN4 instruction:

```
add (4)   r3<4>.xyz:f   r2<4>.yzwx:f   r2.4<4>.zwxy:f   {NoMask}
```

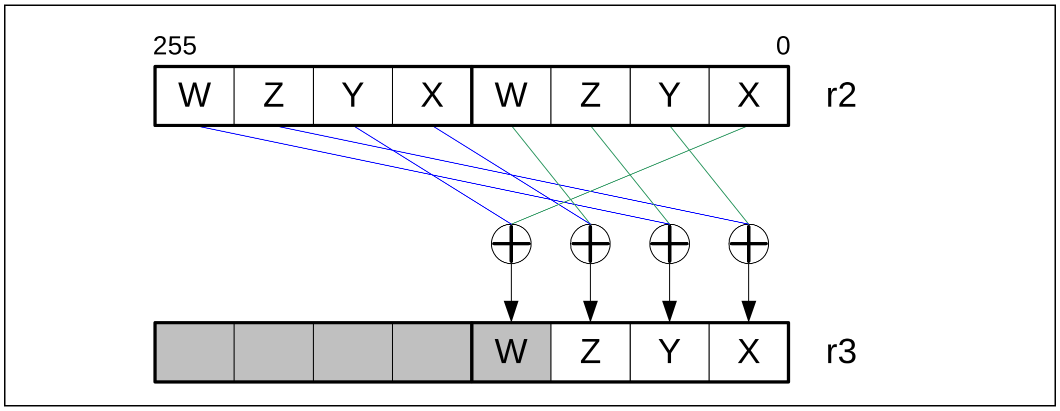
Without diving too much into the syntax definition of a GEN4 instruction, it is clear that a GEN4 instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar to the term SIMD Width used in the literature. Each operand is described by the register region parameters such as '<4>' and data type (e.g. "*:f*"). These will be detailed in Section 11.3. The instruction option field, {NoMask}, ensure that the execution occurs for the execution channels shown in the instruction, instead of, possibly, being masked out by the conditional masks of the thread (See Instruction Summary chapter for definition of *MaskCtrl* instruction field).

The operation of this GEN4 instruction is illustrated in Figure 11-2. In this example, both source operands share the same physical GRF register r2. The two are distinguished by the subregister number. The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 are not modified. In particular, doublewords 4-7 are unchanged as the execution size is 4; doubleword 3 is unchanged due to the destination mask setting.

In this mode of operation, there is only one program flow – any branch decision will be based on a scalar condition and apply to the whole vector of four elements. Option {*NoMask*} ensures that the instruction is not subject to the masks. In fact, most of the instructions in a thread should have {*NoMask*} set.

Even though the execution only performs four parallel add operations, the GEN4 instruction still executes in 2 cycles (with no useful computation in the second cycle).

Figure 11-2. A SIMD4 Example



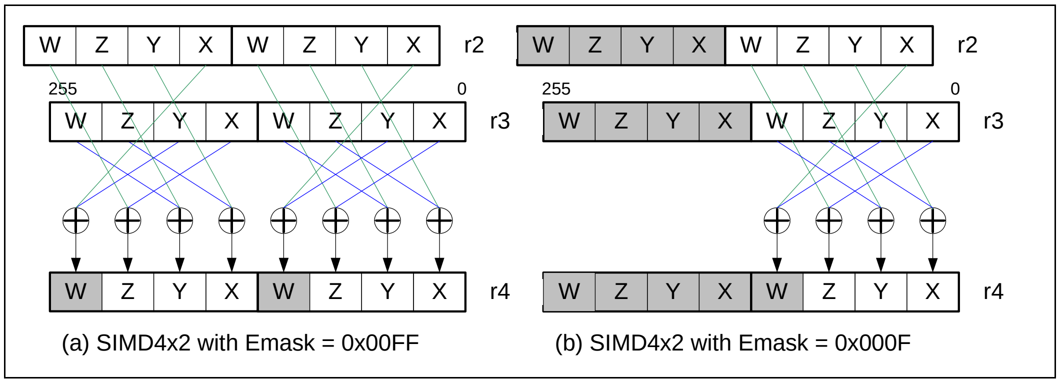
11.2.3 SIMD4x2 Mode of Operation

In this mode, two corresponding vectors from the two program flows fill a GEN4 physical 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 GEN4 instruction:

```
add (8)   r4<4>.xyz:f   r2<4>.yxzw:f   r3<4>.zwxy:f
```

This instruction is subject to the execution mask, which initiated from the dispatch mask. If both program flows are available (e.g. Vertex Shader executed with two active vertices), the dispatch mask is set to 0x00FF. The operation of this GEN4 instruction is illustrated in Figure 11-3 (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 Figure 11-3 (b).

Figure 11-3. SIMD4x2 Examples with Different Emasks

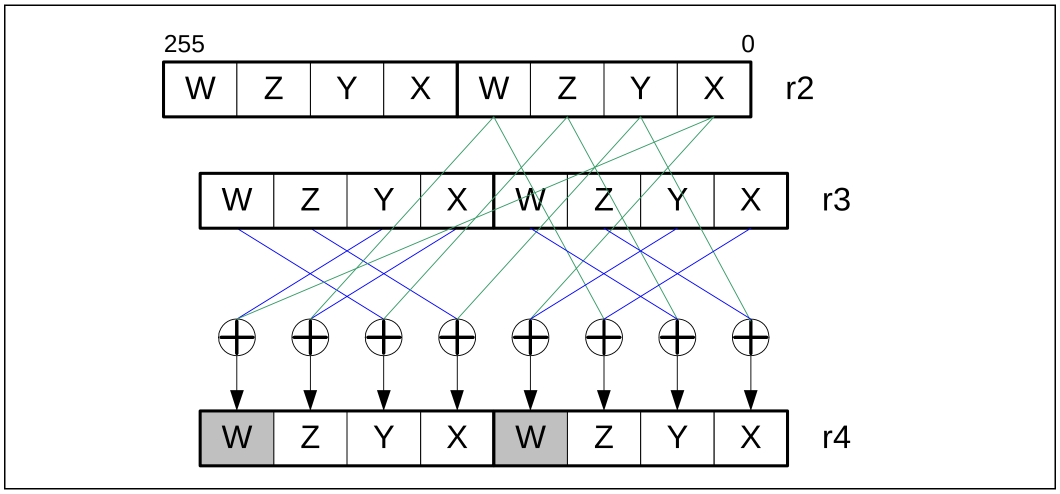


The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2, which is shared by the two program flows. The example 3D graphics API Shader instruction can then be translated into the following GEN4 instruction:

add (8) r4<4>.xyz:f r2<0>.yzwx:f r3<4>.zwxy:f

The only difference here is that the vertical stride of the first source is 0. The operation of this GEN4 instruction is illustrated in Figure 11-4.

Figure 11-4. A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows



11.2.4 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 GEN4 physical 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,*

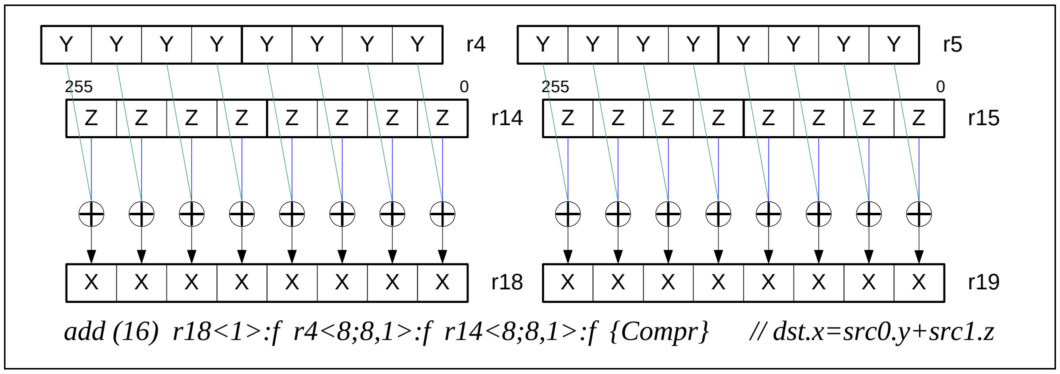
the example 3D graphics API Shader instruction can be translated into the following three GEN4 instructions:

```
add (16)   r18<1>:f   r4<8;8,1>:f   r14<8;8,1>:f   {Compr}      // dst.x = src0.y + src1.z
add (16)   r20<1>:f   r6<8;8,1>:f   r16<8;8,1>:f   {Compr}      // dst.y = src0.z + src1.w
add (16)   r22<1>:f   r8<8;8,1>:f   r10<8;8,1>:f   {Compr}      // dst.z = src0.w + src1.x
```

The three GEN4 instructions correspond to the three enabled destination masks. All instructions are compressed instructions with instruction option of {Compr} (See Instruction Summary chapter for definition of ***ComprCtrl*** field in GEN4 instruction word). All operands are even-aligned GRF registers. 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 Figure 11-5.

With the number of program flows more than one, the above instructions also subject to 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.

Figure 11-5. A SIMD16 Example

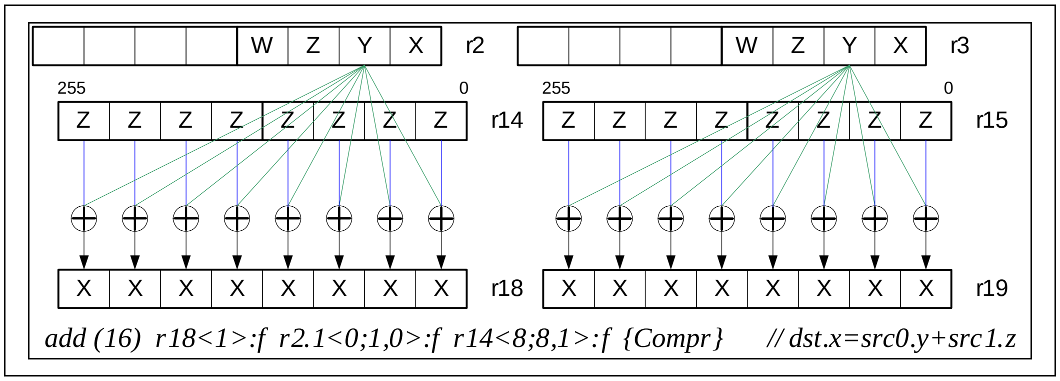


Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS format). The example 3D graphics API Shader instruction can then be translated into the following GEN4 instruction:

```
add (16)   r18<1>:f   r2.1<0;1,0>:f   r14<8;8,1>:f   {Compr}  // dst.x = src0.y + src1.z
add (16)   r20<1>:f   r2.2<0;1,0>:f   r16<8;8,1>:f   {Compr}  // dst.y = src0.z + src1.w
add (16)   r22<1>:f   r2.3<0;1,0>:f   r10<8;8,1>:f   {Compr}  // dst.z = src0.w + src1.x
```

The register region of the first source operand represents a replicated scalar. The operation of the first GEN4 instruction is illustrated in Figure 11-6.

Figure 11-6. Another SIMD16 Example with an AOS Shared Constant


W Z Y X
r2
W Z Y X
r3
255
0
Z Z Z Z Z Z Z Z
r14
r15
X X X X X X X X
r18
r19
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr} // dst.x=src0.y+src1.z

11.2.5 SIMD8 Mode of Operation

Each compressed instruction has two correspond uncompressed instructions. Taking the example instruction shown in Figure 11-6, it is equivalent to the following two instructions.

```
add (8)   r18<1>:f   r4<8;8,1>:f   r14<8;8,1>:f               // dst.x[7:0] = src0.y + src1.z

add (8)   r19<1>:f   r5<8;8,1>:f   r15<8;8,1>:f   {SecHalf}   // dst.x[15:8] = src0.y + src1.z
```

Therefore, SIMD8 can be viewed as a special case for SIMD16.

There are other reasons that SIMD8 instructions may be used. Within a program with 16 concurrent program flows, some time SIMD8 instruction must be used due to architecture restrictions. For example, the address register a0 only have 8 elements, if an indirect GRF addressing is used, SIMD16 instructions are not allowed.

11.3 Registers and Register Regions

11.3.1 Register Files

GEN4 registers are grouped into different name spaces called register files. There are three different register files defined: General Register File, Message Register File, and Architecture Register File. In addition, immediate operands also have a unique encoding of the register file field, even though they come inline in the instruction word and do not have dedicated physical storages.

- General Register File (GRF): GRF contains general-purpose read-write registers.
- Message Register File (MRF): MRF contains special purpose registers used for message passing only. MRF registers are write-only.
- Architecture Register File (ARF): ARF contains all other architectural registers, including the address registers (a#), accumulators (*acc*#), flags (*f*#), masks (*mask*#), mask stack (*ms*#), mask stack depth (*msd*#), notification count (*n*#), instruction pointer (*ip*), etc. Null register (*null*) is also encoded as an ARF register.
- Immediate: Certain instructions take immediate terms as the source operands. These immediate terms have a distinct register file encoding.

Each thread executed in an EU has its own thread context, i.e. dedicated register space, which is not shared between threads executing on a common EU or on a different EU. In the rest of the Chapters, register access are in respect to a given thread.

11.3.2 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
Index-ability:	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, GEN4 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.

Table 11-1. Summary of GRF Registers

Register File	Register Name	Description
General Register File (GRF)	***r#***	General purpose read write registers

Each execution unit has a fixed size physical GRF register RAM. The GRF register RAM is shared by all threads on the EU. GRF space for a thread is allocated at thread dispatch time, allowing the amount of GRF space to adapt to the need of a given thread.

Mapping of a thread's GRF registers to the physical GRF RAM is through a translation table. Therefore, a thread's access to GRF is always through the 0-based logical view. For example, the GRF registers of a thread with 64 GRF register allocation are *r0* through *r63*.

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 physical registers. A destination operand must be contained within one physical 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.

Table 11-2. GRF Registers Available in Device Hardware

Device	Physical Register Size	Allocation Granularity	Number per Thread	Number per EU
[DevBW]	256 bits	16 registers	128 registers	256 registers
[DevCL]	256 bits	16 registers	128 registers	256 registers

11.3.3 MRF Registers

Number of Registers:	Fixed
Default Value:	None
Normal Access:	WO
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Index-ability:	See Table 11-4

Registers in the Message Register File are used to store the header and payload for out-going messages from a thread to a shared function such as the Sampler and Extended Math unit. There are fixed number of MRF registers for each thread.

MRF registers are write-only, and therefore, can only be the destination operand of an instruction.

MRF registers support write-enable at byte granularity. When an MRF register is used as the current destination operand of the *send* instruction, only 256-bit register aligned access is supported.

When accessing (write) outside the MRF register range for a given thread, the result is unpredictable.

Table 11-3. Summary of MRF Registers

Register File	Register Name	Description
Message Register File (MRF)	***m#***	Special purpose output write-only registers

Table 11-4. MRF Registers Available in Device Hardware

Device	Physical Register Size	Number per Thread	Indirect Addressing?
[DevBW]	256 bits	16 registers	No
[DevCL]	256 bits	16 registers	No

Note for Programmers: As a software usage policy, m0 register is reserved for debug. Normal thread should access MRF starting at m1.

11.3.4 ARF Registers

11.3.4.1 Overview

Besides GRF and MRF registers that are directly indicated by unique register file coding, all other registers belong to the general Architecture Register File (ARF). Encoding of architecture register types are based on the MSBs of the register number field, RegNum, in the instruction word. RegNum field has 8 bits. The 4 MSBs, RegNum[7:4], represent the architecture register type. This is summarized in Table 11-5.

Table 11-5. Summary of Architecture Registers

Register Type (RegNum [7:4])	Register Name	Register Count	Description
0000	***null***	*1*	Null register
0001	***a0.#***	*1*	Address register
0010	***acc#***	***2***	Accumulator register
0100	***mask0.#***	*1*	Mask register (active, branch, loop). Note that dispatch mask is RO and in sr#
0101	***ms0.#***	*1*	Mask stack register
0110	***msd0.#***	*1*	Mask stack depth register
0111	***sr0.#***	*1*	State register
1000	***cr0.#***	*1*	Control register
1001	***n#***	*1*	Notification count register
1010	***ip***	*1*	Instruction pointer register
1011-1111	*reserved*		

The remaining register number field RegNum[3:0] is used to identify the register number of a given architecture register type. Therefore, maximum number of registers for a given architecture register type is limited to 16. The subregister number field, SubRegNum, in instruction word has 5 bits. It is used for addressing subregister region for an architecture register supporting register subdivision. SubRegNum field is in unit of byte. Therefore, maximum number of bytes of an architecture register is limited to 32. Depending on alignment restriction of a register type, only certain encodings of SubRegNum field is applicable for an architecture register. The detailed definitions are provided in the following sections.

11.3.4.2 Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF and MRF. The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to control to generate the destination write mask at byte level.

A special restriction on region-based addressing for ARF is that the register region cannot cross register boundary. This rule in fact only applies to the accumulator as it is the only ARF register containing multiple registers (two).

Subregister fields of an ARF register may not all populated (indicated by the subregister indicated as reserved). Write to an unpopulated subregister will be dropped, there is no side effect. Read from an unpopulated subregister, if not specified, will return unpredictable data.

Some of ARF registers are read-only. Write to a read-only ARF register is dropped and there is no side effect.

11.3.4.3 Null Register

ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control:	N/A
Index-ability:	No

The null register is a special encoding for an operand that does not have physical map. It is primarily used in the instruction to indicate the non-existence of an operand. Write to the null register has no side effect. Read from the null register returns undefined result.

The null register can be used in the place when a source operand is absent. For example, for a single source operand instruction such as MOV, 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 physical registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, update to the selected flag register also happens. 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 there is no write completion acknowledgement 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 *send* instruction for details.

11.3.4.4 Address Register

ARF Register Type Encoding (RegNum[7:4]):	0001b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	8
Element Size:	16 bits
Element Type:	UW or UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	N/A
Index-ability:	No

There are eight address subregisters forming an 8-element vector. Each address subregister contains 16 bits. Address subregisters can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the *send* instruction.

Table 11-6. Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***a0*** All other encodings are reserved.	When register a0 or subregisters in a0 is used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned. 00000 = **a0.0:uw** 00010 = **a0.1:uw** 00100 = **a0.2:uw** 00110 = **a0.3:uw** 01000 = **a0.4:uw** 01010 = **a0.5:uw** 01100 = **a0.6:uw** 01110 = **a0.7:uw** All other encodings are reserved. However, when register a0 or subregisters in a0 is an explicit source and/or destination register, other data types are allowed as long as the register region falls in the 128-bit range.

Table 11-7. Address Register Fields

Dword	Bits	Subfield Description
3	31:16	**Address subregister a0.7:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.6:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
2	31:16	**Address subregister a0.5:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.4:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
1	31:16	**Address subregister a0.3:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.2:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
0	31:16	**Address subregister a0.1:uw.** This field can be used for register-indirect register addressing or serve as message descriptor for *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For *send* instruction, it provides the higher 16 bits of <desc>. Format: U12 or U16.
	15:0	**Address subregister a0.0:uw.** This field can be used for register-indirect register addressing or serve as message descriptor for *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For *send* instruction, it provides the lower 16 bits of <desc>. Format: U12 or U16.

When used as a source or destination operand, the address subregisters can be accessed individually or as a group. In the following example, the first instruction moves all 8 address subregisters to the first half of GRF register r1, the second instruction replicates a0.4:uw as an unsigned word to the second half of r1, the third instruction moves the first 4 words in r1 into the first 4 address subregisters, and the fourth instruction replicates r1.4 as a unsigned word to the last 4 address subregisters.

```
mov (8) r1.0<1>:uw a0.0<8;8,1>:uw    // r1.n = a0.n for n = 0 to 7 in words
mov (8) r1.8<1>:uw a0.4<0;1,0>:uw    // r1.m = a0.4 for m = 8 to 15 in words
mov (4) a0.0<1>:uw r1.0<4;4,1>:uw    // a0.n = r1.n for n = 0 to 3 in words
mov (4) a0.4<1>:uw r1.4<0;1,0>:uw    // a0.m = r1.4 for m = 4 to 7 in words
```

When used as the register-indirect addressing for GRF registers, the address subregisters can be accessed also individually or in group. When accessed in group, the address subregisters must be group-aligned. For example, when two address subregisters are used for register indirect addressing, they must be aligned to even address subregisters. In the following example, the first instruction is legal. However, the second one is not. As ExecSize = 8 and the width of <src0> is 4, two address subregisters will be used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address subregisters are a0.2:uw and a0.3:uw. The two align to a dword group in the address register. However, the two address subregisters for the second instruction are a0.3:uw and a0.4:uw. They are not dword aligned in the address register and therefore violate the above mentioned alignment rule.

```
mov (8) r1.0<1>:d r[a0.2]<4,1>:d        // a0.2 and a0.3 is used for src1

mov (8) r1.0<1>:d r[a0.3]<4,1>:d        // ILLEGAL use of register indirect
```

Implementation restriction: *GEN4 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 subregister of a0 register is allowed (i.e. a0.0:ud, which can be viewed as the combination of a0.0:uw and a0.1:uw). In addition, it must be of UD type and in the following form <desc> = a0.0<0;1,0>:ud.*

Implementation restriction: *Elements a0.0 and a0.1 have 16 bits each, but the rest of elements (a0.2:uw through a0.7:uw) only have 12 bits populated each. 12-bit precision supports full indirect-addressing capability for the largest GRF register range. Software must observe the asymmetry of the implementation. When a0.0:uw and a0.1:uw are the source or destination, full 16-bit precision is preserved. However, when a0.2:uw to a0.7:uw are the destination, the higher 4 bits for each element will be dropped; when they are the source, hardware inserts zero to the higher 4 bits for each element.*

Performance Note: *There is only one scoreboard for the whole address register. When a write to some subregisters is in flight, hardware will stall any instruction writing to other subregisters. Software may use the destination dependency control {NoDDChk, NoDDClr} to improve performance in this case. Similarly, when a write to some subregisters is in flight, hardware will stall any instruction sourcing other subregisters until the write retires.*

11.3.4.5 Accumulator Registers

ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	2
Default Value:	None
Normal Access:	RW
Elements:	8 or 16
Element Size:	Various
Element Type:	Various
Access Granularity:	Word
Write Mask Granularity:	N/A
SecHalf Control:	Yes
Index-ability:	No

There are two accumulator registers, *acc0* and *acc1*. They can be accessed either as explicit source and/or destination registers or as implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 doublewords or 16 words of data elements. However, as shown in

Table 11-9, each data element may have higher precision with additional guard bits than that indicated by the numerical data type.

Table 11-8. Register and Subregister Numbers for Accumulate Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***acc0*** 0001 = ***acc1*** All other encodings are reserved.	Reserved: MBZ The accumulator subfields are individually addressable at word granularity. When an accumulator register is an explicit destination, it follows the rules for a destination register. If an accumulator is an explicit source operand, its register region must match with that of the destination register.

The accumulators are implied destination for arithmetic instructions, including parallel and vector instructions. For all other instructions, if accumulator is not specified as the destination operand, the content in the accumulator registers are unaltered. Details can be found in Instruction Reference chapter. There is a control field called Accumulator Disable in control register *cr*0.0 allowing software to turn on (which is the default) and off the implicit update of accumulators.

When an accumulator register is used as an implicit source or destination operand, it is acc0 by default. For a compressed instruction, both acc0 and acc1 are used. If ComprCtrl is set to ***SecHalf***, the implicit accumulator is then acc1. When an accumulator register is used as an explicit source or destination operand, the ***SecHalf*** compression control is ignored. In other words, the implied accumulator (source or destination), if present, is the same as the explicit one.

It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). Result of such instruction is unpredictable.

For a compressed instruction, if an accumulator register is used as an explicit source or destination operand, it must be acc0.

When an accumulator register is used as an explicit source operand, it must be the first source operand <src0>. Meanwhile, source operand modifiers (absolute, negate) are not allowed, as there are ignored by hardware.

When an accumulator register is explicitly or implicitly specified as the destination, destination channel enables do not apply to the accumulator register. In other words, accumulator registers cannot be masked out and the content of the 'disabled' channels of the accumulator register is unpredictable.

When an accumulator register is used as an explicit destination operand, saturation (.sat) is not allowed.

Whether the accumulator register is updated for a given instruction depends on several conditions: it can be an explicit destination operand, it can be an implicit destination for arithmetic and logic instructions and the implicit update is also subject to the control register bit mentioned above. For an instruction in the form like, *opcode <dst> <src0> [<src1>]*, the accumulator register is updated if the any of the following conditions is true

- <dst> is an accumulator register
- *cr0.0[1] is cleared and opcode indicates that the instruction implicitly update accumulator register*

Bit field *cr0.0[1]* is the Accumulator Disable that controls the implied update.

Implementation Restriction due to Floating Point Precision: When a floating point value is stored in the accumulator, it is stored in a non-normalized form with extra precision in mantissa. For an instruction involving addition operation sourcing accumulators, the addition is performed in non-normalized space. Therefore, the results may vary depending on the order of the operations. This is commonly referred to as 'fused' operations. For example, instructions like mac and dp# are fused operation. A group of back-to-back add instructions sourcing accumulators is also a fused operation. Though accumulator may be used as a temporary register with reduced pipeline compute latency, caution must be taken when using accumulator for floating point computation. In general, floating point computation explicitly and/or implicitly involving accumulator should be only used for fused operations where result deviation due to operation order is acceptable. Otherwise, accumulator should not be used as a temporary register.

Errata: *When acc1 is used as explicit operands of two back-to-back instructions, the results may be nondeterministic. This can be worked around using acc1 together with **SecHalf** compression control. This provides almost equivalent behavior except that the second half of the flag register may be used (including when ExecSize = 16). If for certain reason that first half of the flag (or the whole 16-bit of the flag) needs to be used together with acc1, an alternative workaround is to use 'switch' instruction control on the first instruction.*

Performance Note: *GEN4 hardware cannot support write followed by read on the same accumulator register back to back. A thread stall (equivalent of having a 'switch' instruction control) may occur before an instruction that uses an accumulator register as an (implicit or explicit) source and the previous instruction has the same accumulator register as the (implicit or explicit) destination. This commonly occurs in signal processing algorithms. For example, a multi-tap FIR filter can be implemented* *by a sequence of **mac** instructions; a matrix computation in DCT transform also consists of a sequence of **mac** instructions. A program with a non-compressed instruction stream may choose to interleave the use of acc0 and acc1 to achieve better performance, if accumulators are used explicitly. This is not true if the accumulator is addressed implicitly based on the **SecHalf** compression control field as flags and masks are also affected by **SecHalf**.*

***Implementation Precision Restriction:** As there are only 64 bits per channel in dword mode (D and UD), it is sufficient to store 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 results will be 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 multiplication result of two word operands with and without source modifier as the result is up to 33 bits. Integer is 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 risk of overflowing. When overflow occurs, only modular addition can generate 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 UDW (FFFFFFFF) with DW (00000001) results in (1FFFFFFFE). 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.*

Table 11-9. Accumulator Channel Precision

Data Type	# Channel	Bits / Channel	Description
F	8	54+8	When the internal execution data type is float, each accumulator register contains 8 channels of (extended) single precision floating point numbers. The data is in non-normalized format with an 8-bit exponent and a 54-bit mantissa in 2's complement form. The 54-bit mantissa provides 5 extra guide bits over the precision required to store the multiplication result of two 32-bit single precision floats.
D (UD)	8	64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword integer values. The data are always stored in accumulator in 2's complement form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting mul, mach and shr (or mov). [Open: may mention negating a UD may result in unpredictable numbers.]
W (UW)	16	33	When the internal execution data type is doubleword 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 source in W and/or UW format.
B (UB)	N/A	N/A	Not supported data type.

Implementation Restriction about Denorm: *In general, for a floating point arithmetic instruction, hardware converts a denormalized number to sign-preserved zero before performing the computation. However, there is no such a conversion for an accumulator source operand. When accumulator is used as a temporary register for floating point computation, it is software's responsibility to ensure that such conversion is performed when storing floating point data in the accumulator. This is illustrated in the example for the following three-source non-Gen4 'mad' instruction:*

```
mad dst src0 src1 src2  // dst = src0 + src1 * src2
```

One might intuitively consider translating this into two Gen4 instructions: a mov to accumulator followed by a mac:

```
mov acc0.0:f src0:f// acc0.0 = src0 (this is a 'raw' move)

mac dst src1 src2   // dst = acc0.0 + src1 * src2
```

This may generate incorrect floating results as the first move instruction doesn't flush denorms in src0 into zeros in acc0.0. Consequently, these denorm numbers in accumulator may create undefined results for the mac instruction. A correct translation should use an instruction that forces the denorm-to-zero flush, such as a multiplication of 1 or an addition of negative zero as show below. Yes, it must be a negative zero to preserve the sign of the source.

```
add acc0.0:f src0:f -0.0:f   // acc0.0 = src0 (with denorm-to-zero flush)

mac dst src1 src2   // dst = acc0.0 + src1 * src2
```

11.3.4.6 Flag Register

ARF Register Type Encoding (RegNum[7:4]):	0011b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	2
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	Yes
Index-ability:	No

There is one flag register that consists of two 16-bit subregisters. Each flag subregister can be individually addressed. Each bit of a flag subregister corresponds to a data channel. (See Table 11-10 for details). Furthermore, each 16-bit subregister may be split to half when ***ComprCtrl*** field is set to ***SecHalf*** in the instruction.

The two flag subregisters (*f0.0:uw* and *f0.1:uw*) can be used as the destination of the conditional modifier and can also be the source of the predication. As both predication flag source and conditional flag destination share the same instruction field, when both are enabled, they use the same flag subregister.

The values held in the individual bits of a flag subregister are the result of the most recent instruction which performed a condition-code evaluation with that flag register specified as the destination of the evaluation result. For example "*add.nz.f0.0 ...*" updates flag f0.0 with the per-channel results of the not-zero condition. The flag subregister has per-bit write enables. When being updated as the secondary destination associated with conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. The other bits in the flag subregister are unchanged.

The flag register as a whole or as two subregisters can also be an explicit source and/or destination operand.

Table 11-10. Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***f0*** All other encodings are reserved.	00000 = **f0.0:uw** 00010 = **f0.1:uw** All other encodings are reserved.

Table 11-11. Flag Register Fields

Dword	Bits	Subfield Description
0	31:16	**Flag subregister f0.1:uw.** This field contains 16 bits of conditional flags. It can be used for predication and branch instructions. This field can be updated as the regular destination operand of an instruction or as the secondary destination operands associated with conditional modifier. This field can serve as regular source operand of an instruction or serve as the source for predication used for regular instructions or branch instructions. Format: U16
	15:0	**Flag subregister f0.0:uw** Same as f0.1:uw. Format: U16

11.3.4.7 Mask Registers

ARF Register Type Encoding (RegNum[7:4]):	0100b
Number of Registers:	1
Default Value:	DMask
Normal Access:	RW
Elements:	4
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	Yes
Index-ability:	No

There is one mask register that contains four subfields with 16-bit each. Each 16-bit mask subfield can be split into two halfs (8 bits each). The four mask subregisters, namely, *amask*, *imask*, *lmask* and *cmask*, are used to form the *emask* to control the channel enables for SIMD instructions. They can be updated using the branch instructions such as *if*, *else*, *endif*, *do*, *while*, *break*, etc. A value one at a bit location of a mask subfield indicates that the channel is active with respect to the mask subfield. A value zero at a bit location indicates that the channel is inactive with respect to the mask subfield. For example, an *if* instruction may change some bits in *imask* from one to zero, effectively turning these channels off for the if-endif-block. In the mean time, the previous value in imask is pushed in the mask stack register. The corresponding *endif* instruction pops the mask stack register and restores the *imask* value, effectively re-enabled the channels turned off by the *if* instruction.

The four mask subfields can be accessed individually or as a group using region based register addressing. Explicit access of the second half of the mask subfield is through the compression control of ***SecHalf***.

Table 11-12. Register and Subregister Numbers for Mask Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***mask0*** All other encodings are reserved.	00000 = **mask0.0:uw (amask).** It contains the active mask register 00010 = **mask0.1:uw (imask).** It contains the if-mask register 00100 = **mask0.2:uw (lmask).** It contains the loop-mask register 00110 = **mask0.3:uw (cmask).** It contains the continue-mask register All other encodings are reserved.

Table 11-13. Mask Register Fields

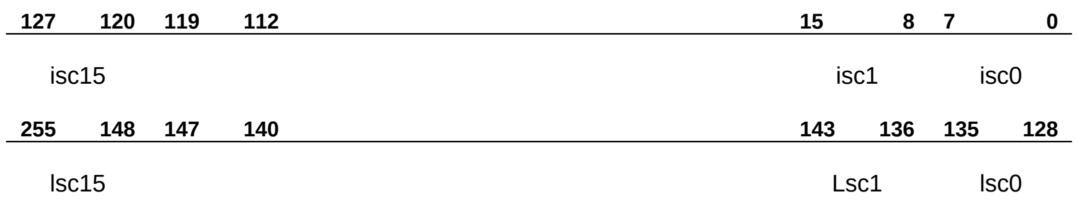
Dword	Bits	Subfield Description
1	31:16	**Continue-Mask (mask0.3:uw or cmask:uw).** This field contains the 16-bit continue mask. Format: U16
	15:0	**Loop-Mask (mask0.2:uw or lmask:uw).** This field contains the 16-bit loop mask. Format: U16
0	31:16	**If-Mask (mask0.1:uw or imask:uw).** This field contains the 16-bit if-mask. Format: U16
	15:0	**Active-Mask (mask0.0:uw or amask:uw).** This field contains the 16-bit active mask. Format: U16

Implementation Restriction on Register Access: *When a mask register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to '**switch**' for an instruction that uses mask registers as an explicit operand.*

Relaxed Restriction on Register Access*: Software must ensure that there is a delay of at least four clock cycles between an instruction, A, that explicitly read the mask register and another instruction, B, that implicitly writes the mask register. This may be achieved, for example, by setting the thread control field to '****switch'*** *for instruction A or by inserting two nop instructions between instructions A and B. Note that one GEN4 instruction, at minimal, takes two clock cycles.*

11.3.4.8 Mask Stack Register

ARF Register Type Encoding (RegNum[7:4]):	0101b
Number of Registers:	1
Default Value:	0 (See Erratum below)
Normal Access:	RW
Elements:	32
Element Size:	8 bits
Element Type:	UB
Access Granularity:	DQ
Write Mask Granularity:	DQ
SecHalf Control:	No
Index-ability:	No



The Mask Stack register consists of two stacks: *istack* and *lstack*, 128-bit each. The If-Stack, *istack*, is for 'if' conditional code blocks. The Loop Stack, *lstack*, is for looping code blocks. Unlike traditional hardware stacks, the Mask Stack is implemented by counters, with one counter per execution channel. There are 16 bytes in each stack, corresponding to the 16 counters. The lower 4 bits within each byte is the counter value. The upper 4 bits is forced to zero when the Mask Stack register is the source and are ignored when the Mask Stack register is the destination.

The counters in the mask stack register are initialized to zero at thread dispatch. When a mask is pushed to a mask stack (e.g. an *if* instruction pushes *imask* to *istack*), the counters in the mask stack subfield are updated based on the following rules:

- If a bit of the mask is one, the corresponding mask stack counter is not changed
- If a bit of the mask is zero, the corresponding mask stack counter is incremented by one

When a mask is restored from popping a mask stack (e.g. an *endif* instruction pops *istack* and restores *imask*), the counters in the mask stack subfield are updated based on the following rules:

- If a mask stack counter is zero, it remains zero; and a one is written to the corresponding bit location of the mask
- If a mask stack counter is not zero, it is decremented by one; and a zero is written to the corresponding bit location of the mask

The hardware mask stack is provided to support nested loops/branches. The premise of a counter based mask stack is that once an execution channel is disabled (the mask bit is set to zero, for example, by a *do* instruction), it will remain disabled for subsequent nested branches. It can only be re-enabled after all nested loops/branches within a loop/branch block are completed (the mask bit is restored to one, for example, by the matching *while* instruction to the above mentioned *do* instruction). Therefore, only the number of nesting for a disabled channel needed to be stored in the mask stack. The counter in the mask stack register counts this number.

The bit layout of the Mask Stack register is shown in Figure 11-7. *IStack* takes the lower 128 bits and *LStack* is in the upper 128 bits. Each 128-bit subfield should not be further subdivided.

When SPF bit in *cr0* register is set, this register becomes read-only – contents cannot be modified implicitly or explicitly.

Table 11-14. Register and Subregister Numbers for Mask Stack Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***ms0*** All other encodings are reserved.	00000 = **ms0.0:ub (istack).** It contains the if-stack register 10000 = **ms0.16:ub (lstack).** It contains the loop-stack register All other encodings are reserved.

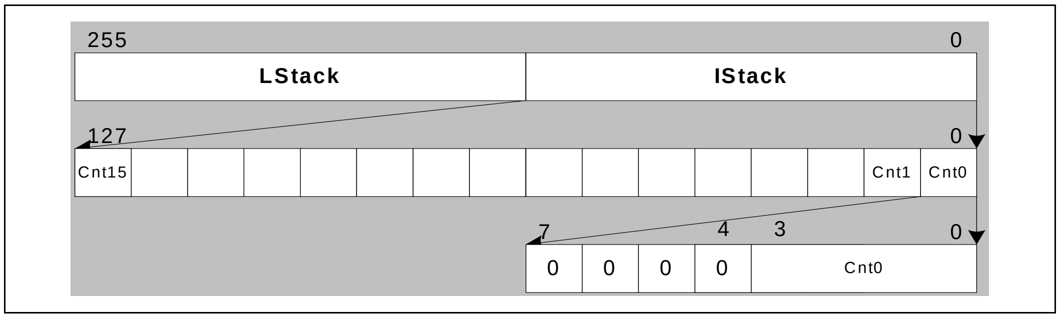
Table 11-15. Mask Stack Register Fields

Subregister	Bits	Subfield Description
isc# **(# = 0 to 15)**	7:4	Reserved: MBZ.
	3:0	**If-Stack Count #:** This is the 4-bit If-stack count value for execution channel #, counting the number of 1's has pushed into the If-stack. The 4-bit count per channel provides maximum nesting depth of 15. For nesting beyond 15 levels deep, software must manually save/restore the Mask Stack register. Format: U4
lsc# **(# = 0 to 15)**	7:4	Reserved: MBZ.
	3:0	**Loop Stack Count #:** This is the 4-bit loop stack count value for execution channel #, counting the number of 1's has pushed into the loop stack. The 4-bit count per channel provides maximum nesting depth of 7 as one *loop*-instruction pushes both *cmask* and *bmask* in the same mask stack. For nesting beyond 7 levels deep, software must manually save/restore the MaskStack register. Format: U4

The IStack, LStack, or the Mask Stack register, as a whole, can be an explicit source or destination operand. Smaller grain accesses to the register are not allowed. For example, the stacks can be saved/restored using 'mov' instruction via GRF and/or MRF registers as shown in the next instruction.

```
mov (16) r5.0<1>:w ms:w    // save mask stack to r5
    …                      // other instructions
    mov (16) ms:w r5.0<1>:w   // restore mask stack from r5
```

Figure 11-7. Format of the Mask Stack Register



Implementation Restriction on Register Access: *When a mask stack register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to '**switch'** for an instruction that uses mask registers as an explicit operand.*

This register access restriction is not applicable, hardware does ensure execution pipeline coherency, when a mask stack register is used as an explicit source and/or destination.

[DevBW, DevCL] Erratum: The subfields in mask stack register are reset to zero during graphics reset, however, they are not initialized at thread dispatch. These subfields will retain the values from the previous thread. Software should make sure the mask stack is empty (reset to zero) before terminating the thread. In case that this is not practical, software may have to reset the mask stack at the beginning of each kernel, which will impact the performance.

11.3.4.9 Mask Stack Depth Register

ARF Register Type Encoding (RegNum[7:4]):	0110b
Number of Registers:	1
Default Value:	0 (See erratum below)
Normal Access:	R/W
Elements:	2
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	No
Index-ability:	No

The Mask Stack Depth register, *msd0*, allows a software view of the current depth of the Mask Stack. Depths for both the IStack and the LStack are maintained in a single 32-bit register, with individual depths accessible using region-based register addressing. Each depth value occupies a word. Read and write access is provided to allow for extending the Mask Stack Depth beyond 7 or 15 through manual stack management.

Both depths are initialized to 0 at thread-load time. When SPF bit in *cr0* register is set, this register becomes read-only – contents cannot be modified implicitly or explicitly.

The depth value is incremented by hardware each time a value is pushed to the associated Mask Stack. The value is decremented each time a value is popped from the Mask Stack. The addition/subtraction operates in modular math on the 4-bit unsigned integer fields. Whenever, the Mask Stack Depth value **exceeds** the overflow/underflow trigger values, a mask stack overflow or underflow exception will be generated. Upon a mask stack underflow/overflow exception, software may manually update the mask stack and mask stack depth registers based on the amount of underflow/overflow.

Table 11-16. Overflow / Underflow Exception Trigger Value

Device	Underflow	Overflow
[DevBW]	0	15
[DevCL]	0	7

For example, if the current depth value is 15 and the overflow trigger value is 15, an instruction that pushes two masks into a mask stack will cause mask stack overflow exception and the resulting depth value is 1. Upon an overflow exception, software may compute the amount of overflow by adding 1 to the depth value in the register. In this case, the amount of overflow is 2.

Another example, if the current depth value is 5 (0101b) and the underflow trigger value is 0, an instruction that pops a mask stack register by 9 (1001b) will cause mask stack underflow exception with a resulting depth value of 12 (1100b). By subtracting the depth value from 16, software can determine the amount of underflow. In this case, it is 4.

Implementation Restrictions*: When the Overflow trigger value equals to the maximum mask stack value, the overflow exception mechanism is not sufficient to support extended stack managed by software. This is because that software must reset the Mask Stack Depth value for the overflowed mask stack to 0 and reset the mask stack values to 0 or 1. If the original value of a channel is 0, it remains 0; if the original value of a channel is non-zero, it is reset to 1. Continuing pushing into the stack will eventually cause the mask stack counter to wrap before the Mask Stack Depth value to exceed the overflow trigger value. This will cause functional error. Software may choose to use different workarounds for the device that has the overflow trigger value equals to the maximum mask stack value. Here are a few workaround examples:*

- *For loops/branch within the hardware limit, software can use the hardware stack*
- *For loops/branch beyond the limit, there are several solutions*
  - *Use single program flow - processing a pixel at a time. May not be a benchmark or real performance issue*
  - *Use software jitted stack management (without using the overflow exception mechanism)*
- *For example, software may manually manage the overflow cases, ensuring that overflow exception will never be reached. This solution may use the underflow exception though.*

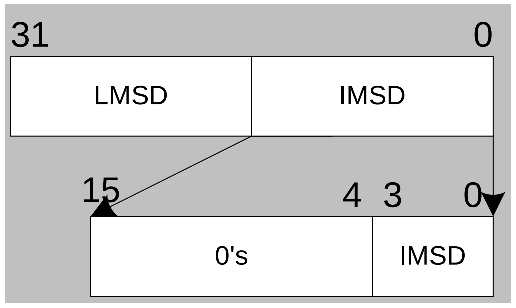
Table 11-17. Register and Subregister Numbers for Mask Stack Depth Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***msd0*** All other encodings are reserved.	00000 = **msd0.0:w (imsd).** It contains the if-stack depth register 00010 = **msd0.1:w (lmsd).** It contains the loop-stack depth register All other encodings are reserved.

Table 11-18. Mask Stack Depth Register Fields

DWord	Bits	Subfield Description
0 (lmsd)	31:20	Reserved: MBZ.
	19:16	**Loop-Stack Depth:** Current depth of the LStack. The maximum allowed nesting of *loop*-instructions is half of the overflow trigger value before a stack overflow or underflow. This is because that loop-instructions such as do (or while/break/cont) push (or pop) both lmask and cmask to (or from) the LStack. When overflow/underflow occurs, this field contains the modular residue of the amount of overflow/underflow. Initialized to 0 at thread load time. Format: U4
1 (imsd)	15:4	Reserved: MBZ.
	3:0	**If-Stack Depth:** Current depth of the IStack. The maximum allowed nesting of *if*-instructions equals to the overflow trigger value before a stack overflow or underflows. When overflow/underflow occurs, this field contains the modular residue of the amount of overflow/underflow. Initialized to 0 at thread load time. Format: U4

Figure 11-8. Format of the Mask Stack Depth Register



Implementation Restriction on Register Access: *When a mask stack depth register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to '**switch'** for an instruction that uses mask registers as an explicit operand.*

This register access restriction is not applicable, hardware does ensure execution pipeline coherency, when a mask stack depth register is used as an explicit source and/or destination.

Erratum: The subfields in mask stack depth register are reset to zero during graphics reset, however, they are not initialized at thread dispatch. These subfields will retain the values from the previous thread. Software should make sure the mask stack depth values are zero before terminating the thread. In case that this is not practical, software may have to reset the mask stack depth at the beginning of each kernel, which will impact the performance.

HW stack overflow and underflow exception are not supported for both IStack and LStack. Software has to maintain stack depth to ensure stack will not be overflowed and underflowed. Both stack overflow in HW when they reach 16 for BWR and 7 for CLN, and underflow when they pass 0.

11.3.4.10 State Registers

ARF Register Type Encoding (RegNum[7:4]):	0111b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RO
Elements:	2
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Byte
Write Mask Granularity:	N/A
SecHalf Control:	No
Index-ability:	No

Thread state registers are read-only registers. They can be accessed as a whole or individually using region-based addressing with byte granularity.

Table 11-19. Register and Subregister Numbers for State Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***sr0*** All other encodings are reserved.	Valid encoding range: 00000 – 00111 (in unit of byte) For example, 00000 = **sr0.0:ud.** It contains general thread states such as EUID/TID 00100 = **sr0.1:ud.** It contains the dispatch mask All other encodings are reserved.

Table 11-20. State Register Fields

DWord	Bits	Subfield Description
0 (sr0.0:ud)	31:28	Reserved: MBZ
	27:24	**FFID** (Fixed Function Identifier). Specifies which fixed function unit generates the current thread. This field is set at thread dispatch and is forwarded on the message bus for all out-going messages from this thread.
	23:19	Reserved: MBZ
	18	**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 shared function chapters whether and how a shared function uses this priority hint. 0 = Low priority class 1 = High priority class
	17:16	**Priority.** This field is the relative aging priority of the thread. This field indicates the 'age' of this thread relative to other thread within the EU. No two threads within 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, dispatching a new thread to this EU will cause the old thread's priority number being incremented to 1. However, if the active thread (assuming for simplicity there is only one) on an EU has a priority number 1 (or 2 or 3), dispatching a new thread to this EU will not change the old thread's priority number. As threads on an EU may terminate in arbitrary orders, the exact number for a thread depends on the dynamic execution of threads.
	15:12	Reserved: MBZ
	11:8	**EUID** (Execution Unit Identifier). Specifies which execution unit the current thread is at.
	7:3	Reserved: MBZ
	2:0	**TID** (The thread identifier). Specifies which thread slot the current thread is assigned to. This field is set at thread dispatch.
1 (sr0.1:ud)	31:24	Reserved: MBZ
	23:20	**GRF Register Blocks.** This field contains a number of GRF register count in unit of 16 registers. The valid range is from 16 to 128 GRF registers. 0000: Reserved 0001 – 1000: 16 GRF registers to 128 GRF registers 1001 – 1111: Reserved
	19:16	Reserved : MBZ

DWord	Bits	Subfield Description
	15:0	**Dispatch Mask** (**DMask**). The 16-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register. Format: U16

11.3.4.11 Control Register

ARF Register Type Encoding (RegNum[7:4]):	1000b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	4
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

The Control register is a read-write register. It contains four 32-bit subregisters that can be accessed individually.

Subregister *cr0.0*:*ud* contains normal operation control fields such as the floating point mode and the accumulator disable. It also contains the master exception status/control field that allows software to switch back to the application thread from the system routine. Debug control field Breakpoint Suppress is also in *cr*0.0:*ud*.

Subregister *cr0.1*:*ud* contains the mask and status/control fields for all exceptions. The exception fields are arranged in significance-decreasing order from MSB to LSB. This allows the system routine to use *lzd* instruction to find the high priority exceptions and handles them first. As each exception is mapped to a single bit, other exception priority order may be implemented by software. System routine may choose to handle one exception at a time, by handle the exception detected by a *lzd* instruction and return to application thread. Or it may choose to handle all the concurrent exceptions, by looping through the exception fields until all outstanding exceptions are handled before returning back to the application thread.

Exception enable bits (bits 15:0 in *cr0.1*:*ud*) control whether an exception will cause hardware to jump to system routine or not. Exception status and control bits (bits 31:16 in *cr*0.1:*ud*) indicate which exceptions have occurred and are used for system routine to clear the exception. Even if a given exception is disabled, the corresponding exception status and control bit still reflects the status whether an exception event has occurred or not.

cr0.2:*ud* contains the **Application IP (AIP)** indicating the current thread IP when an exception occurs.

cr0.3:*ud* is reserved. Writing to this subregister is dropped; result of reading from this subregister is unpredictable.

Fields in Control registers also refer to a virtual register called **System IP (SIP)**. SIP is the virtual register holding the global System IP, which is the initial instruction pointer for the system routine. There is only one SIP for the whole system. It is virtual only from a thread’s point of view, as it is not visible (i.e. not readable and not writeable) to the thread software executed on a GEN4 EU. It can only be accessed indirectly by the hardware to response to exception events. Upon an exception, hardware performs some book keepings (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 *cr*0, which will cause the hardware to load AIP to IP register. See STATE_SIP command for how to set SIP.

Details about exception handling and debug can be found in the debugging and exceptions chapters.

Table 11-21. Register and Subregister Numbers for Control Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***cr0***	00000 = **cr0.0:ud.** It contains general thread control fields
All other encodings are reserved.	00100 = **cr0.1:ud.** It contains exception status and control
	01000 = **cr0.2:ud.** It contains AIP.10100 **(reserved)**
	All other encodings are reserved.

Table 11-22. Control Register Fields

DWord	Bits	Subfield Description
0 (cr0.0:ud)	31	**Master Exception State and Control.** This field 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 field to 1 and switch to SIP. Writing a 1 to this field has no effect. Writing a 0 to this field also has no effect if the previous value is 0. In both cases, the field keeps the previous value. If the previous value of this field 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 field then stays as 0. This field is initialized to 0. 0 = Indicate that the thread is in normal state 1 = Indicate that the thread is in exception state
	30:16	Reserved: MBZ

DWord	Bits	Subfield Description
	15	**Breakpoint Suppress.** This field specifies whether breakpoint exception is suppressed or not. This field is normally set by software and cleared by hardware. If Master Exception Status and Control field is 1, this field is ignored by hardware. If Master Exception Status and Control field is 0 (i.e. not in system routine) and Breakpoint is enabled: If this field is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This field is provided to prevent infinite loop 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 field) before returning to the application thread. This field has no effect when Breakpoint Enable bits is cleared. This field is initialized to 0. 0 = Breakpoint exception is not suppressed 1 = Breakpoint exception is suppressed
	14:9	Reserved : MBZ
	8	**Host Notification Data.** This field is forwarded to the debug MMIO registers (EU Debug Register 4 and 5 – Attention Data registers) when a host notification signal is sent by the EU upon executing instruction "WAIT n1". Combining host notification signaling mechanism, this data field provides a one-way communication interface from the thread to the host. Even though it may be slow, this thread/host interface is independent of the message bus structure and any shared function. Note that this communication interface does not allow the host to return data to the thread. More details can be found in Debugging chapter. This field is initialized to 0.
	7:3	Reserved : MBZ
	2	**Single Program Flow (SPF).** Specifies whether the thread has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). This field affects the operation of all branch instructions. If SPF is not set, branch instructions such as *if*, *do*, *while*, push the current branch masks into the corresponding mask stack registers or pop the mask stack and updates the corresponding masks. Use of the mask stack supports concurrent execution of multiple program flows in a single thread using SIMD instructions. If SPF is set, the mask stack push or pop actions are inhibited for these branch instructions. Therefore, the top of stack will not be modified by any branch instruction. In Single Program Flow mode, all execution channels branch and/or loop identically. By hold the mask stack unchanged for all branch instructions, infinite level of nesting of branch instructions can be supported. This field is initialized by the Thread Dispatch. 0 = Multiple Program Flows 1 = Single Program Flow

DWord	Bits	Subfield Description
	1	**Accumulator Disable.** This field controls the implicit update of the accumulator. If this field is cleared, the accumulator is updated for all instructions in which it is designated implicitly. If set, the accumulator is disabled for all implicit update operations, maintaining its value prior to being disabled. Setting this field has no effect if the accumulator is the explicit destination operand for an instruction. This field 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 which include implicit update of the accumulator. In order to use those instructions within the System Routine, but still preserving the accumulator contents upon return 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 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 prior to 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 would be required to preserve the content of this bit.
	0	**Floating Point Mode (FPMode).** This field specifies whether the current floating point operation mode is in IEEE standard mode or the alternative mode. It is used to control the floating operation of the Execution Unit. It is also forwarded on the message sideband for all out-going messages, for example, to control the floating point mode of the Extended Math unit or the Sampler unit. Software may modify this field to dynamically switch between the two floating point modes. This field is initialized by the Thread Dispatch. 0 = IEEE floating point mode 1 = Alternative floating point mode

DWord	Bits	Subfield Description
1 (cr0.1:ud) [31:16] StatCtlr [15:0] Masking	31	**Breakpoint Exception Status and Control.** This field, when set, indicates breakpoint exception. Under breakpoint condition, hardware sets this bit upon entering the system routine. This field is used for kernel debug in both breakpoint and single stepping modes. For normal breakpoint handle, the system (debug) routine should reset this field and also set the Breakpoint Suppress field before returning back to the application thread. If this bit is not reset in the system routine, upon returning to application routine, hardware executes one instruction (if Breakpoint Suppress field was set in the system routine), and then jumps to system routine again. Therefore, by not clearing this field, single stepping can be emulated. This bit may be set and cleared by software. This field is initialized to 0. There is a restriction associated with this field when emulating single stepping debug on – it does not work on compressed instructions. When system routine sets this field as well as the Breadkpoint Suppress field for a compressed instruction before returning to application routine, the instruction pointer is not automatically advanced by hardware. This prohibits the intended single-stepping mechanism supplied by hardware. Two software workarounds are available, which emulate the single stepping debugging capability with certain drawbacks. 1. The host debugger software may expose to the debug user interface with breakpoint and single stepping methods but set the breakpoint for all instructions in kernel binary. For breakpoint events that don't match with any user exposed breakpoint/single-stepping instructions, the host debugger software may silently skip the false breakpoint and simply let the system routine to return back to application routine. This approach has low performance. 2. Alternatively, the host debugger software may set/clean the breakpoint field on an instruction by instruction basis to emulate single stepping. After servicing a breakpoint and before letting the system routine to return to the application routine, the host debugger may perform the following steps: flush the instruction cache, restore the breakpoint field for the current instruction, store the breakpoint field for the next instruction, and then set the breakpoint field for the next instruction. This process is repeated each step of source code stepping. A side effect is that the kernel instructions in memory are dynamically modified, resulting other threads under debug (with breakpoint enabled) to cause breakpoint exception on an unintended instruction. Host debugger software must detect and skip these false exceptions (e.g. based on identifier of the thread).
	30	**External Halt Exception Status and Control.** This field indicates the External Halt exception. It is set by EU hardware upon receiving the broadcast External Halt signal. System routine should reset this field before returning to application routine in order to avoid infinite loop. This bit may be set or cleared by software. This field is initialized to 0.

DWord	Bits	Subfield Description
	29	**Software Exception Control.** This is the control field of software exception. Setting this field to 1 in application routine will cause an exception. Clearing this field in application routine has no effect. Upon entering system routine, the hardware maintains this field as one to signify software exception. System routine should reset this field before returning to application routine. This field may be set or cleared by software. This field is initialized to 0.
	28	**Illegal Opcode Exception Status.** This field, when set, indicates illegal opcode exception. The exception handle 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 application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	27	**LStack Overflow Exception Status.** This field, when set, indicates the LStack overflow exception. System routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	26	**LStack Underflow Exception Status and Control.** This field, when set, indicates the LStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	25	**IStack Overflow Exception Status and Control.** This field, when set, indicates the IStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread.Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	24	**IStack Underflow Exception Status and Control.** This field, when set, indicates the IStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	23	Reserved: MBZ.
	22:16	Reserved: MBZ

DWord	Bits	Subfield Description
	15	**Breakpoint Enable.** Specifies whether breakpoint exception is enabled or not. This field is initialized by the Thread Dispatcher. Format = ENABLED 0 = Disabled 1 = Enabled
	14	**External Halt Exception Enable.** This field specifies whether the External Halt Exception is enabled or not. When this bit is set, the thread allows to be interrupted by an external halt signal. The usage of this exception is for a host debug software to halt the execution of threads in GEN4 EUs at any selected time. With proper debug handling in SIP, execution may resume normally after thread being halted. This field is initialized by the Thread Dispatcher. Format = ENABLED
	13	**Software Exception Enable.** This field enables or disables the software exception. Enabling or disabling this field may allow host software to turn on/off certain features (such as profiling) without changing the kernel program. This field is initialized by the Thread Dispatcher. Format = ENABLED
	12	**Illegal Opcode Exception Enable.** This field specifies whether illegal opcode exception is enabled or not. Illegal opcode exception includes illegal opcode and undefined opcode, caused by bad program or run time data corruption. This field is initialized by the Thread Dispatcher. Software should normally set it in the interface descriptor. Even though the mechanism is provided to disable illegal opcode exception, it should be used with extreme caution. Format = ENABLED
	11	**MaskStack Exception Enable.** This field specifies whether MaskStack Exceptions are enabled or not. It is used to control all four MaskStack Exceptions (LStack Overflow and Underflow, IStack Overflow and Underflow). This field is initialized by the MaskStack Exception Enable bit from the Thread Dispatcher. Software should normally set it in the interface descriptor. Even though the mechanism is provided to disable stack overflow/underflow exception, it should be used with extreme caution. For the kernel that is known to not overflow/underflow mask stacks, enable this exception has no adverse effect. Format = ENABLED **Erratum:** This field is reserved: MBZ.
	10	Reserved: MBZ
	9:0	Reserved : MBZ

DWord	Bits	Subfield Description
2 (cr0.2:ud)	31:4	**Application IP (AIP).** This is the register storing the instruction pointer before an exception is handled. Upon an exception, hardware automatically saves the current IP into the AIP register, and then sets the **Master Exception State and Control** field to 1, which forces a switch to the System IP (SIP). AIP register may contain either the pointer to the instruction that causes the exception (such as breakpoint for debug), or the one after (such as mask stack overflow/underflow exceptions). This is shown in the following table, where IP is the instruction which generated the exception. **Exception Type** — **AIP Value** Breakpoint — IP External Halt — n/a (1) Software Exception — IP + 1 Illegal Opcode — IP Mask Stack Overflow / Underflow — IP + 1 (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 returning IP to be altered after an exception handle.
	3:0	Reserved : MBZ

***Implementation Restriction on Register Access:** When the control register is used as an explicit source and/or destination, hardware doesn't 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 indeterministic results.*

11.3.4.12 Notification Registers

ARF Register Type Encoding (RegNum[7:4]):	1001b
Number of Registers:	3
Default Value:	No
Normal Access:	RO
Elements:	3
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

There are two notification registers (*n0:ud* and *n1:ud*) used by the *wait* instruction. These registers are read-only and can be accessed in 32-bit granularity.

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximal allowable 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. That means if there are multiple host write to EU Debug Register 6 or 7 (Attention Clear registers) may be collapsed into a single notification event to the target thread, if only one wait instruction is executed during that time period.

When directly accessed, this register is read-only. If the value is none zero, the only way to alter the value is to use the wait instruction to decrement the value until zero is reach. A wait instruction on a zero notification subregister will cause the thread to stall, waiting for a notification signal from outside targeting to the same subregister. See wait instruction for details.

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

Table 11-23. Register and Subregister Numbers for Notification Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***n0*** 0001 = ***n1***All other encodings are reserved.	Reserved.

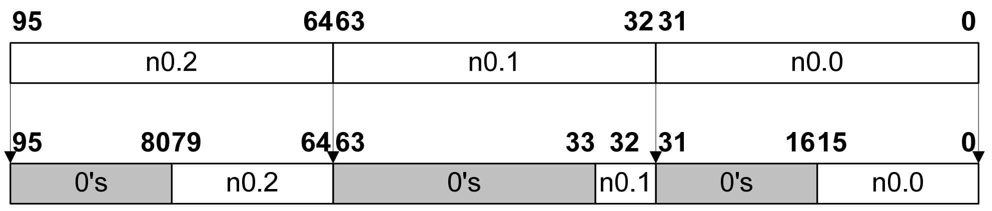
Table 11-24. Fields of Notification Register n0

DWord	Bits	Subfield Description
0	31:7	Reserved: MBZ
	6:0	**Thread to Thread Notification Count.** This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U7

Table 11-25. Fields of Notification Register n1

DWord	Bits	Subfield Description
0	31:1	Reserved : MBZ
	0	**Host to Thread Notification.** This register is used by the WAIT instruction for host-to-thread synchronization via MMIO registers EU Debug Register 6 and 7 (Attention Clear registers). See Debugging chapter for details. Format: U1
	15:0	**Thread to Thread Notification Count.** This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U16

Table 11-26. Format of the Notification Register



11.3.4.13 IP Register

ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

The ip register can be accessed as a 32-bit quantity. It is a read-write register, containing the current instruction pointer, which is relative to the **Generate State Base Address**. Reading this register returns the instruction pointer of the current instruction. The 3 LSBs are always read as zero. Writing this register forces the program flow to jump to the new address. When it is written, the 3 LSBs are dropped by hardware.

Table 11-27. Register and Subregister Numbers for IP Register

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

Table 11-28. IP Register Fields

DWord	Bits	Subfield Description
0	31:4	**Ip.** Specifies the current instruction pointer. This pointer is relative to the **General State Base Address**.
	3:0	Reserved : MBZ
	2:0	Reserved : MBZ

11.3.5 Immediate

Two forms of immediate are provided as a source operand: scalar and vector.

For a scalar immediate, it can be of any of the specified numerical 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 numerical types are reserved for future extensions.

The immediate field in a GEN4 instruction has 32 bits as shown below. 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 GEN4 instruction.

31	**0**
imm32	

31	**0**
must be the same as [15:0]	**imm16**

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 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 in 2's complement form (halfbyte). Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following table.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
immV7	immV6	immV5	immV4	immV3	ummV2	immV1	immV0

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. Each 8-bit subfield contains a signed float with 3-bit exponent and 4-bit fraction. Each 8-bit subfield provides signed floating point values with restricted range and precision. This is depicted in the following table.

31 24	23 16	15 8	7 0
immVF3	immVF1	immVF1	immVF0

*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**).*

11.3.6 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 GEN4 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 GEN4. 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 Figure 11-9 and Figure 11-10. As shown in Figure 11-9, 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 GEN4 GRF register file, the outer loop may be unrolled for GEN4. 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 GEN4 to combine data conditioning operations with computing operations. These techniques in GEN4 architecture help to achieve high compute efficiency and throughput for graphics and media applications.

Figure 11-9. Conventional SIMD Instruction Sequence

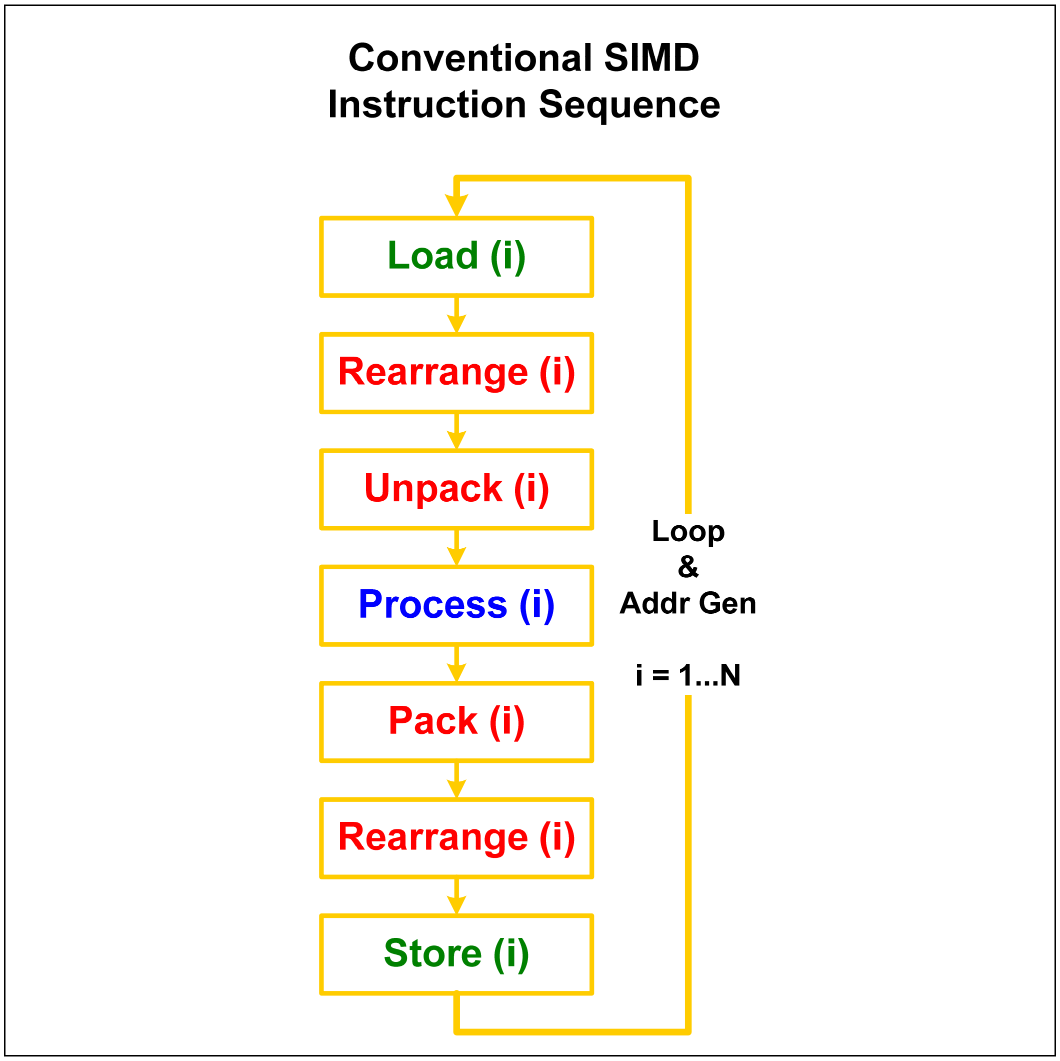
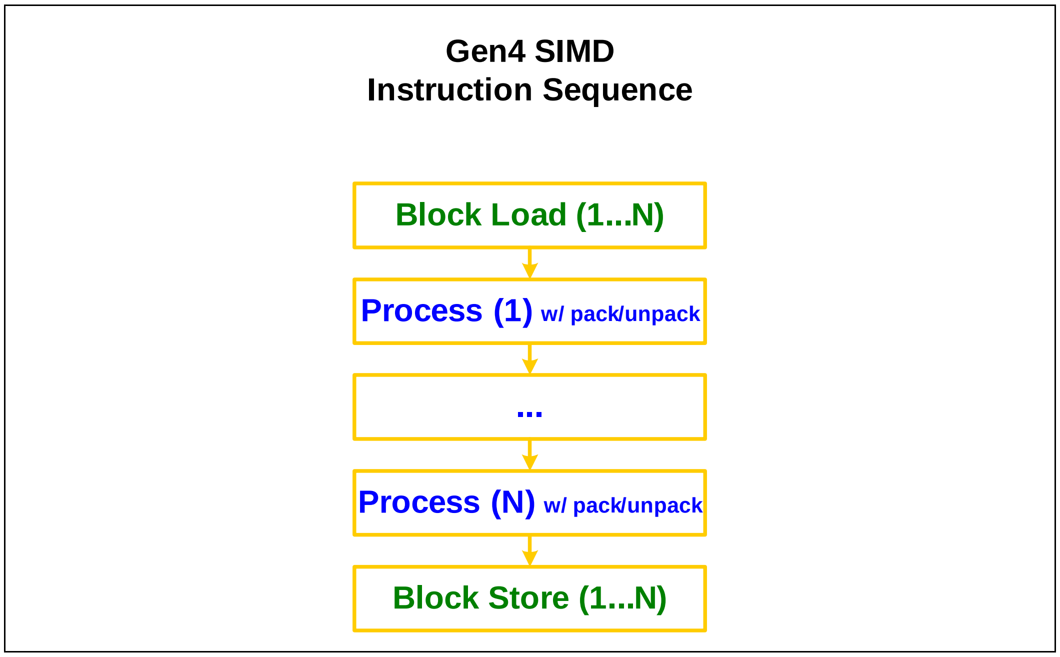


Figure 11-10. GEN4 SIMD Instruction Sequence for the Same Program



In a GEN4 instruction, each operand defines a region in the register file. A region may contain multiple data elements. Each data element is assigned to an execution channel in the EU. The total number of data elements of a region is called the **size** of the region, or the size of the operand. The number of execution channels is called the **execution size** (***ExecSize***), which is specified in the instruction word. ExecSize determines the size of region for source and destination operands in an instruction.

- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
  - Exception of this rule is present for the integer reduction instructions (such as sad2 and sada2) where the destination area is smaller than the source area.

Regions are **generalized 2-dimensional** (2D) arrays in row-major order. The first dimension is named the **horizontal** dimension (data elements within a row) and the second dimension is termed the **vertical** dimension (data elements in a column). Here, horizontal/vertical and row/column are just symbolic notations. When the GRF or MRF 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 11.3.4.6.

Specifically, a region-based description of a source operand can take the following format

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 subregister 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 numerical data type (float, word, byte, etc.) of the data elements. All data elements within a region must have the same type.

In GEN4, both GRF and MRF register files consist of a sequence of 256-bit physical registers. When viewing the register file (GRF for example) as a sequence of 256-bit aligned physical registers, *RegNum* field provides the physical register number, thus for the name. *SubRegNum* provides the sub-field addressing within a physical 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.

Figure 11-11 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.

Figure 11-11. An example of a register region (*r4.1<16;8,2>:w*) with 16 elements

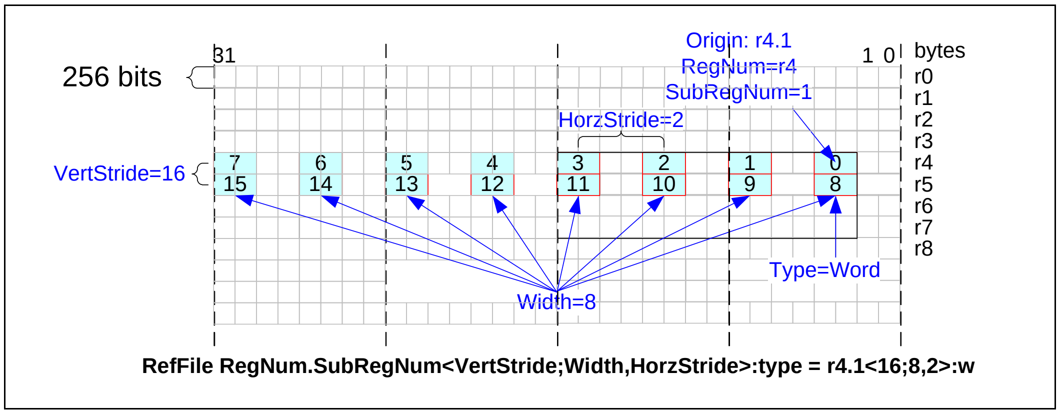
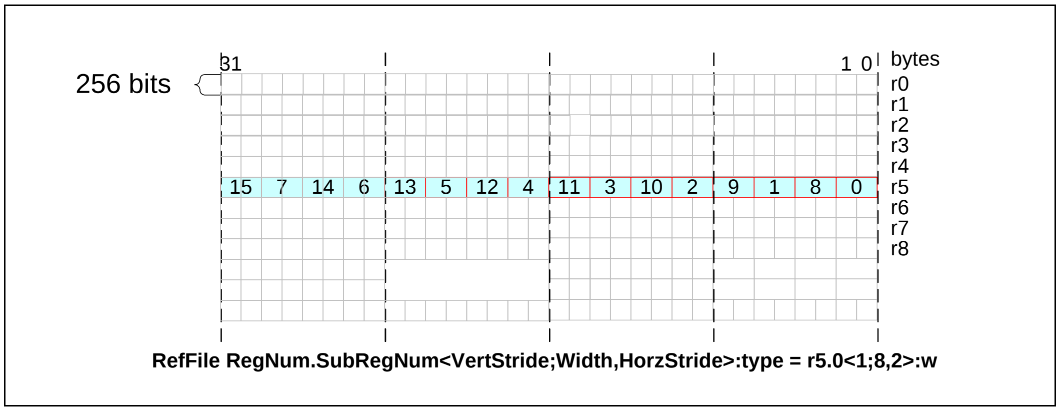


Figure 11-12 shows another example where the rows are interleaved. The region, having word data elements, starts at location r5.0:w. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location 5.2:w. And there are 8 elements per row. VertStride, the distance 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.

Figure 11-12. A 16-element register region with interleaved rows (*r5.0<1;8,2>:w*)


256 bits
31
1 0
bytes
r0
r1
r2
r3
r4
r5
r6
r7
r8
15 7 14 6 13 5 12 4 11 3 10 2 9 1 8 0
RefFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type = 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 across multiple physical GRF register, 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 physical registers are not supported.

11.3.7 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.

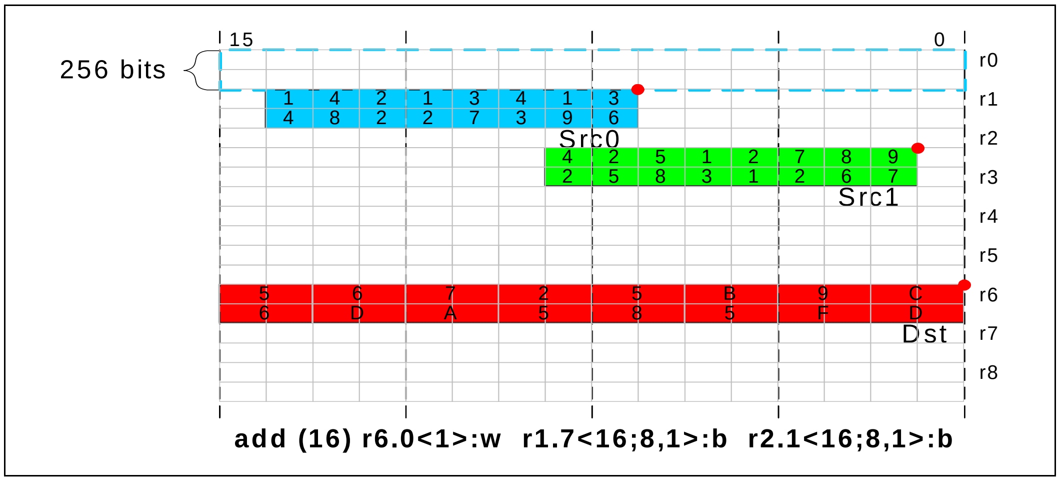
11.3.7.1 Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Figure 11-13 and Figure 11-14 are two examples of direct register addressing.

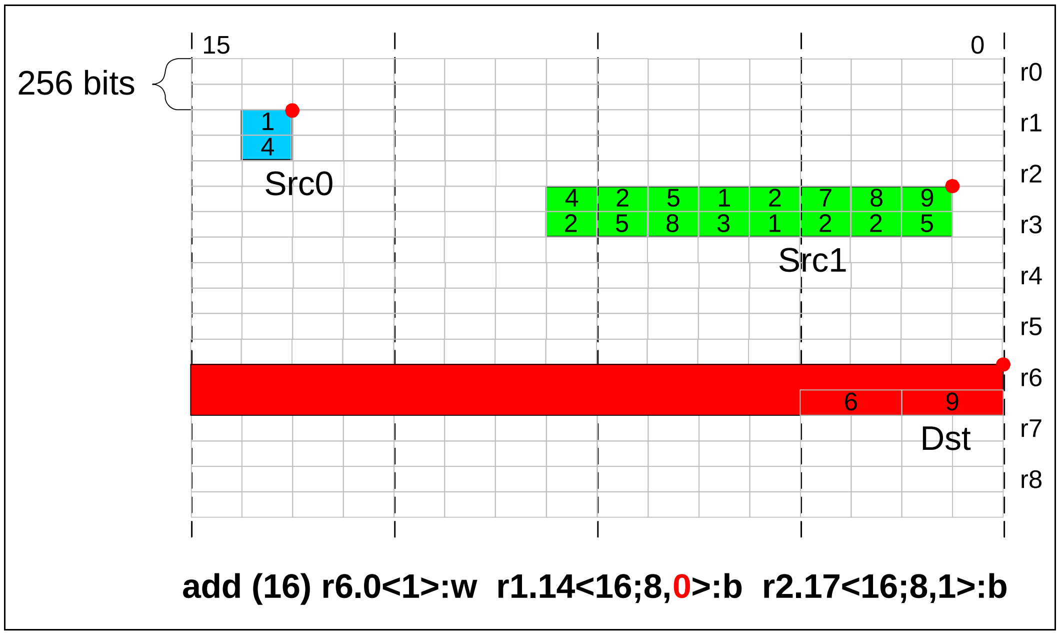
For the example in Figure 11-13, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical stride of 16 bytes for *Src0* and *Src1* indicates that the first element and the 9'th element are 16 bytes apart in the register file. Note that *Src0* falls into the 256-bit physical GRF register starting at r1.0, but Src1 crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have C = 3+9.

Figure 11-13. A region description example in direct register addressing



For the example in Figure 11-14, the sizes of areas of *Src0* and *Src1* are the same, but *Src0* contains a vector of replicated scalars. With HorzStride = 0 and Width = 8, the first row of 8 elements in Src0 is a replication of the byte at r1.14. Comparing *ExecSize* of 16 to Width of 8 indicates that there is a second row of 8 elements in *Src0*. With VertStride = 16, the second row in *Src0* is a replication of the byte at r1.20 (20 = 14+16). Effectively, the 16 data elements of *Src0* are {1,1,1,1,1,1,1,1, 4,4,4,4,4,4,4,4}.

Figure 11-14. A region description example in direct register addressing with <src0> as a vector of replicated scalars

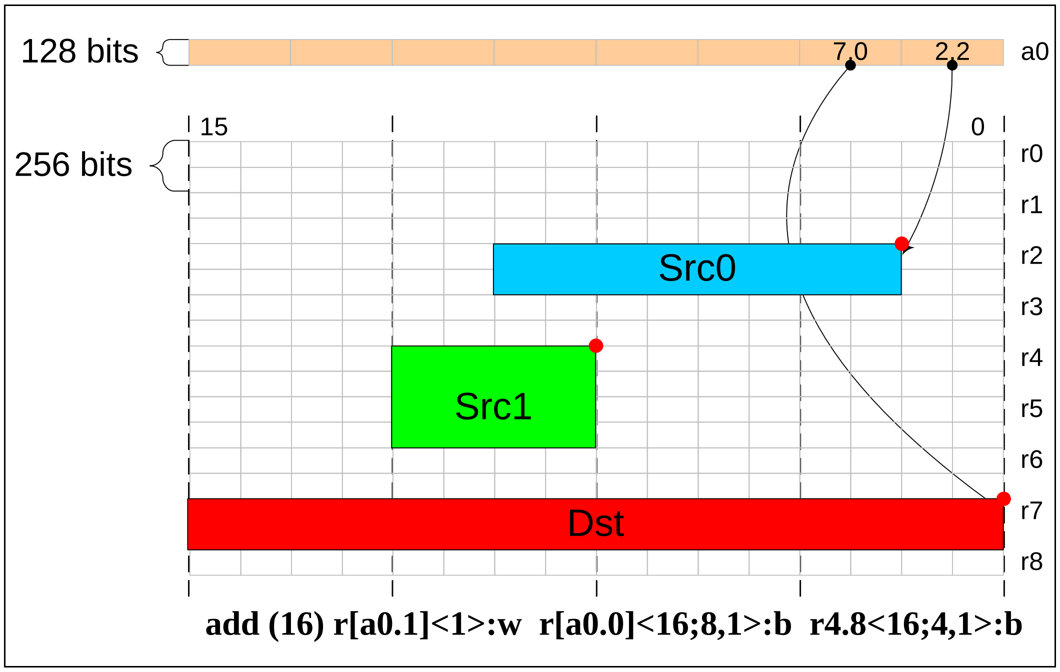


11.3.7.2 Register-indirect Register Addressing with a 1x1 Index Region

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.

Figure 11-15 depicts an example for this addressing mode. For example, the present of full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.

Figure 11-15. An example illustrating register-indirect register addressing mode with a 1x1 index region

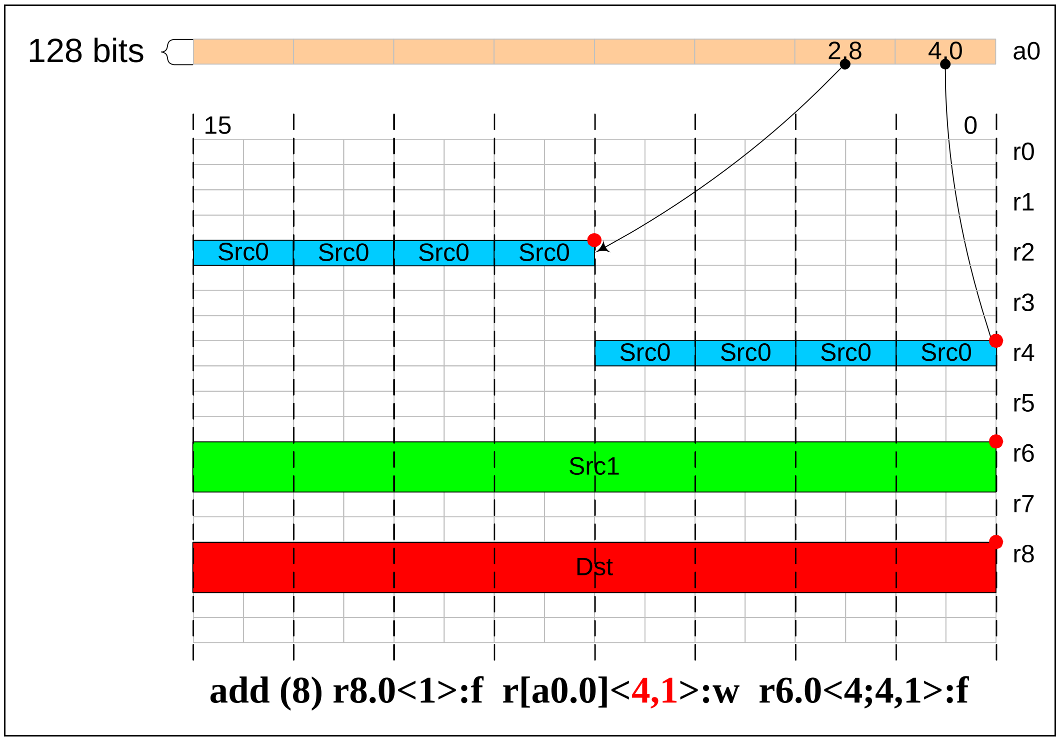


11.3.7.3 Register-indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, 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, the following contiguous address registers are for the following rows. The total number of address registers used is inferred by parameters *ExecSize* and *Width.*

An example is provided in Figure 11-16. The assembly syntax notion of a register region without vertical stride, <4,1>, corresponding to the special encoding of vertical stride of 0xF in the instruction word, indicates the VxH or Vx1 mode of indirect register addressing. In this case, the origin for each row of Src0 is provided by the address register. As ExecSize/Width = 2, there are two address registers a0.0 and a0.1, each pointing to a row of 4 data elements.

Figure 11-16. An example illustrating register-indirect-register addressing mode with a Vx1 index region (Src0)

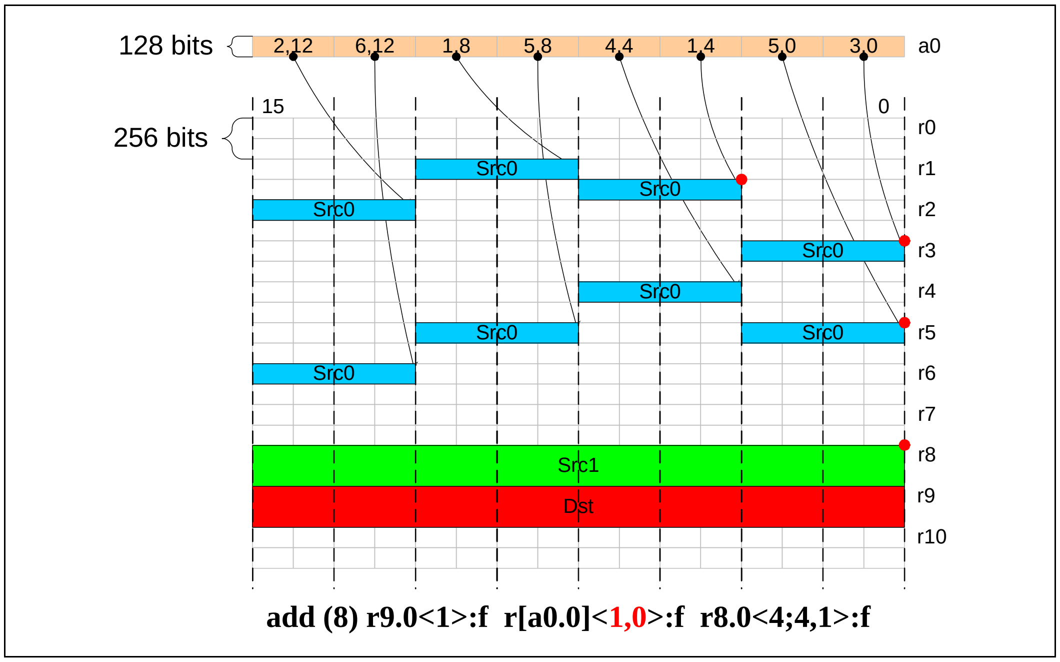


11.3.7.4 Register-indirect Register Addressing with a VxH Index Region

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 Figure 11-17. 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.

Figure 11-17. An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



11.3.8 Access Modes

There are two basic GEN4 register access modes controlled by a single bit instruction subfield called Access Mode.

- 16-byte Aligned Access Mode (**align16**): In this mode, the origins of all operands (sources and destination), whether it is by direct addressing or register-indirect addressing, are 16-byte aligned. For example a row in the region description starts at 16-bype aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
  — The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.

- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region register descriptions are supported, however, source swizzle or destination mask are not supported.
  — The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.

11.3.9 Execution Data Type

GEN4 architecture supports instructions with mixed data types. The internal hardware computation is performed using the execution data type. When an instruction has only one source operand or has two source operands of the same data type, the execution data type is the same as that of the source. When an instruction has two source operands of different types, an execution data type is determined and one of the source operands will be converted to the execution type before the computation is performed. The execution type is independent of the destination data type. When the destination data type is different from the execution data type, a type conversion is performed on the intermediate compute results before the results are written into the destination register. Such a destination type conversion doesn't apply to accumulator registers, implicitly or explicitly. Therefore, accumulator type cannot differ from the execution data type.

Determination of the execution data type for two sources of different data types obeys the following rules

- If any source is a float, the execution data type is float (F)
- Else if any source is a dword, the execution data type is signed dword integer (D)
- Else execution data type is signed word integer (W)

Note that when the execution data type is an integer, it is always a signed integer. This doesn't affect the functional correctness of the instruction as extra precisions are carried within the hardware, including the accumulator. See Instruction Reference Chapter for detailed description for each instruction.

11.3.10 Register Region Restrictions

The following register region rules apply to the GEN4 implementation. Rules and restrictions for compressed instructions can be found in the Instruction Compression section.

1. *ExecSize* must be equal to or less than the maximum execution size supported for the operand type. As shown in Table 11-29, the maximum execution size is determined by the largest operand type of the sources and destination of the instruction.
2. 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, plus *ExecSize* and destination region description.
3. *ExecSize* must be equal to or greater than *Width*.
4. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.
5. If *ExecSize* = *Width* but *HorzStride* = 0, there is no restriction on *VertStride*.
6. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
7. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be set to zero.
8. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.

9. Destination region cannot cross the 256-bit register boundary.
    9.1. Exception to this rule is for a compressed instruction where the destination region covers exactly **two adjacent** 256-bit physical registers.
10. Destination region alignment rule.
    10.1. With the exception on 'raw move' described in rule #10.3 and the exception on byte destination in rule #10.5, all destination data elements must be aligned to the size for the execution data type of the instruction. For example, if one of the source operands is in dword mode (a float, a signed or unsigned dword integer), the execution data type will be either float or signed dword integer. Therefore, the destination data elements must be dword aligned. This rule has the following two implications:
        10.1.1. The destination sub-register must be aligned to the size of the execution data type.
        10.1.2. If *ExecSize* is greater than 1, *dst.HorzStride**sizeof*(dst.Type)* must be equal to or greater than the size of the execution data type.
    10.2. If *ExecSize* is 1, *dst.HorzStride* must not be 0. Note that this is relaxed from rule 10.1.2. Also note that this rule for destination horizontal stride is different from that for source as stated in rule #7.
    10.3. When destination type is byte (UB or B), only a 'raw move' using ***mov*** instruction supports packed byte destination register region: *dst.HorzStride* = 1 and *dst.type* = (UB or B). This packed byte destination region is not allowed for any other instructions, including a 'raw move' using ***sel*** instruction. This is because ***sel*** instruction is based on word or dword wide execution channels.
    10.4. When an instruction has a source region that spans two physical registers, one of the followings must be true:
        10.4.1. Destination region is entirely contained in the lower oword of a physical register,
        10.4.2. Destination region is entirely contained in the upper oword of a physical register, or
        10.4.3. Destination elements are evenly split between the two owords of a physical register.
    10.5. Relaxed alignment rule for byte destination. When destination type is byte (UB or B), destination data elements can be either aligned to 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 the following two implications:
        10.5.1. The destination sub-register must be either aligned to the size of the execution data type or one byte higher off the execution data type.
        10.5.2. If *ExecSize* is greater than 1, *dst.HorzStride**sizeof*(dst.Type)* must be equal to or greater than the size of the execution data type. This is the same as that in #10.1.2.
11. In Align1 access mode, a source region must be within **two adjacent** 256-bit physical registers.
    11.1. It is further restricted that a single row cannot cross physical register boundary.
    11.2. It is further restricted that for source 1 only, when crossing physical register boundary, the vertical stride must be equivalent to 32-byte.

12. In Align16 access mode, a source region must be within **one** 256-bit physical registers.
13. Rules on register-indirect register access:
    13.1. Source 0, source 1 and destination can be indexed. Support for indirect register access for source operands is device dependent. See Table 11-30 for details.

    13.2. If supported, an indexed source 1 can only have a 1x1 indexed register region – only single index mode is allowed for a source 1.

    13.3. An indexed destination can only have a 1x1 indexed register region – only single index mode is allowed for a destination operand.

    13.4. Data elements referenced by a single index within a source region cannot cross 256-bit physical register boundary. This applies to register region with a single index or with multiple indices.

        13.4.1. A register region with multiple indices may access multiple physical registers as long as data elements associated with each index follow the above-mentioned rule. For example instruction "mov (16) r0.0:uw r[a0.0]<2,2>:uw" is allowed. This is a source gathering instruction whereas the source operand may potentially tough 8 different physical GRF registers.

    13.5. VxH index regioning using 2 index registers cannot start from a0.6. a0.6 can still be used as part of the VxH index regioning as long as the start index register is not itself, a0.6 can also be used in Vx1 index regioning by itself.

    13.6. **[DevCL]** When indirect addressing is used on a source or destination of an instruction, the following combination for the particular source or destination is not allowed.

        13.6.1. using indirect addressing

        13.6.2. the index register used is a0.7

        13.6.3. the index immediate is negative

        13.6.4. the datatype is byte

14. ***Implementation Restriction:*** Any non-compressed instruction with 2row regioning source(s) cannot be a jump/branch target or follows directly after a conditional branch instruction or a predicated jmpi instruction, unless the 2 rows are previously written from FPU, not from outside of EU. The workaround would be either break the instruction into 2 1row instruction, or insert a NOP.

Implementation Restriction: *The relaxed alignment rule for byte destination (#10.5) is not supported.*

Table 11-29. Execution size in device hardware

Device	Native GEN4 Instructions			Compressed GEN4 Instructions		
Max Operand Size	DWORD	WORD	BYTE	DWORD	WORD	BYTE
[DevBW]	8	16	16	16	32	32
[DevCL]	8	16	16	16	32	32

Table 11-30. Indirect source addressing support available in device hardware

Device	Indirect Source 0	Indirect Source 1
[DevBW]	Yes	No
[DevCL]	Yes	Yes

Note: It is expected that some of the restrictions may be relaxed in future implementations of the GEN4 architecture.

11.3.10.1 Examples

Some examples are provided here to illustrate the cases when the register region restrictions are violated. It is provided as informative material to help understanding these restrictions.

Example 1: The following instructions are illegal as they violate rule #10.1, as the destination is not aligned to the execution data type.

```
mov (1) r0.1<1>:b r2.0:w  // dst.SubReg must be even
mov (2) r0.0<1>:b r2.0:w  // dst.HorzStride must be >= 2
mov (2) r0.0<2>:b r2.0:d  // dst.HorzStride must be >= 4
mov (2) r0.0<2>:b r2.0:f  // dst.HorzStride must be >= 4
mov (1) r0.2<1>:b r2.0:d  // dst.SubReg must be dword aligned
```

Example 2: This instruction is illegal as it violates rule #10.1.2, as when *ExecSize* = 1, *dst.HorzStride* cannot be zero.

```
mov(1) r0.0<0>:b r0.0:d
```

Example 3: This instruction is illegal as it violates rule #11.1, as the source contains one row of 2 elements that spans physical register r2 and r3.

```
mov (2)  r1.0:d r2.7<2;2;1>:d
```

Example 4: This instruction is illegal as it violates rule #14, as the jump target of the jmpi instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)jmpi L0
...
L0: mov (8) r10 r4<8;4,1>  <- source0 uses both r4 and r5
```

Example 5: This instruction is illegal as it violates rule #14, as the instruction immediately after the predicated jmpi instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)jmpi L0
mov (8) r10 r4<8;4,1>  <- source0 uses both r4 and r5
```

Example 6: This instruction is illegal as it violates rule #14, as the jump target of the iff instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)iff L0
...
endif
L0: mov (8) r10 r4<8;4,1>  <- source0 uses both r4 and r5
```

Example 7: This instruction is illegal as it violates rule #14, as the instruction immediately follows the if instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)if L0
mov (8) r10 r4<8;4,1>  <- source0 uses both r4 and r5
...
L0: else
```

11.3.10.2 Different Raw Moves

Definition of Raw Move: Raw move is an operation that moves data elements from source to destination without altering the bit fields of the data elements. It must use one of the move instructions such as ***mov, sel, movi***. Arithmetic instruction that results in unaltered bit fields of the data elements are not treated as raw move. A raw move may subject to the execution channel enables by using prediction or being present in multi-channel branch code segment. Type conversion by definition cannot be used in a raw move. Therefore, source and destination operands must be of the identical data type. For example, if both source and destination are float, for an arithmetic instruction, denorm will be flushed to zero. However, for a raw move, denorm will be preserved.

Definition of Byte Raw Move: As the minimal execution channel type is word, when the destination stride is greater than one byte, each data element of the source can be mapped to one execution channel. This is referred to as Byte Raw Move. Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. Byte Raw move subjects to execution channel enables.

Definition of Packed-Byte Raw Move: As the minimal execution channel type is word, when the destination stride is equal to one byte, two data elements of the source are mapped to one execution channel. This is referred to as Packed-Byte Raw Move. Packed-Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. However, as the data elements are not mapped to execution channels, undefined results may occur if Packed-Byte Raw Move is mixed with execution channel enables. So for Packed-Byte Raw Move, ***NoMask*** should be used when there are un-enabled channels within the execution size of the instruction.

11.3.11 Destination Operand Description

11.3.11.1 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

11.4 SIMD Execution Control

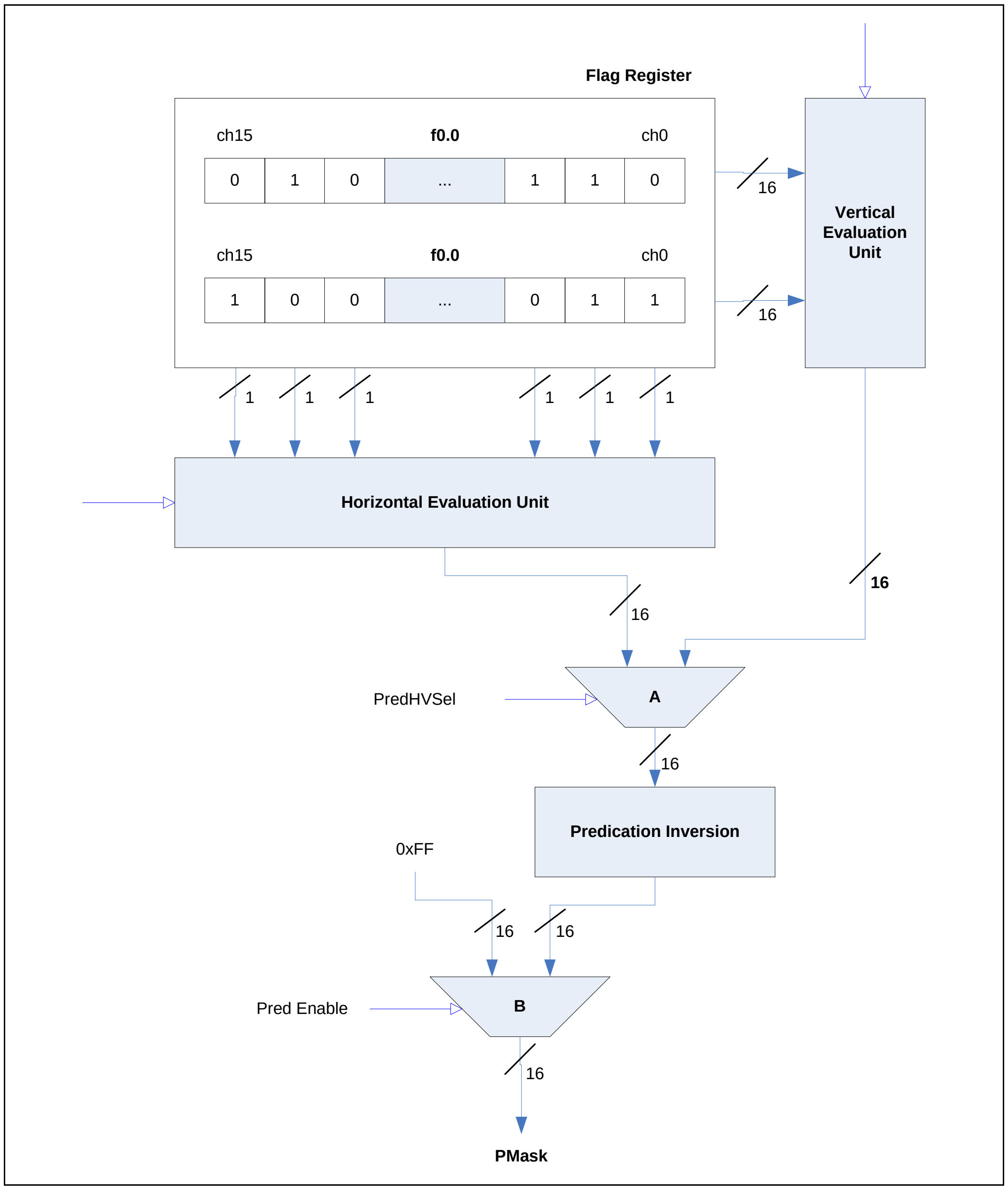
11.4.1 Predication

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.

Figure 11-18 shows the block diagram of the hardware logic to generate PMask. PMask is generated based on combinatory logic operation of the bits in the flag register. Instruction field *PredCtrl* controls the horizontal evaluation unit and vertical evaluation unit. MUX A in the figure selects whether horizontally-evaluated results or vertically-evaluated results are sent to the Predication Invertion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.

Figure 11-18. Generation of predication mask



11.4.2 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 Figure 11-18. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.

11.4.3 Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is 'no combination' – the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```
If (PredCtrl == "Sequential flag channel mapping") {
    For (ch=0; ch<16; ch++)
        PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}
```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of '.x', '.y', '.z' and '.w' select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = '.x' means that channel 0 in each group is replicated.
- OR combination: PredCtrl of '.any4h' means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of '.all4h' means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```
If (Access Mode == Align16) {
    For (ch = 0; ch < 16; ch += 4)
        Switch (PredCtrl) {
        Case '.x':   bTmp = f0.#[ch]; break;
        Case '.y':   bTmp = f0.#[ch+1]; break;
        Case '.z':   bTmp = f0.#[ch+2]; break;
        Case '.w':       bTmp = f0.#[ch+3]; break;
        Case '.any4h':   bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
        Case '.all4h':   bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
                         }
                         bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
                         PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
            }
}
```

In **Align1** access mode, horizontal combination is based on AND combination '.any#h' and OR combination '.all#h' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.

```
If (Access Mode == Align1) {
    Switch (PredCtrl) {
        Case '.any2h':   groupSize = 2; <op> = '|'; break;
        Case '.all2h':   groupSize = 2; <op> = '&'; break;
        Case '.any4h':   groupSize = 4; <op> = '|'; break;
        Case '.all4h':   groupSize = 4; <op> = '&'; break;
        Case '.any8h':   groupSize = 8; <op> = '|'; break;
        Case '.all8h':   groupSize = 8; <op> = '&'; break;
        Case '.any16h':  groupSize = 16; <op> = '|'; break;
        Case '.all16h':  groupSize = 16; <op> = '&'; break;
    }
    For (ch = 0; ch < 16; ch += groupSize) {
        For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)
                                bTmp = bTmp <op> f0.#[ch+inc];
                    For (inc = 0; inc < groupSize; inc ++)
            PMask[ch+inc] = bTmp;
            }
}
```

11.4.4 Predication with Vertical Combination

Predication with vertical combination uses both flag subregister as inputs. The AND or OR combination is across the subregisters on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {
    For (ch = 0; ch < 16; ch ++) {
        If (PredCtrl == 'any2v')
            PMask[ch] = f0.0[ch] | f0.1[ch]
        Else If (PredCtrl == 'any2h')
            PMask[ch] = f0.0[ch] & f0.1[ch]
    }
}
```

11.5 Instruction Compression

11.5.1 Motivation and Expected Usage

When pixel processing is performed in the channel serial mode, each 3D graphics API Pixel Shader instruction (SIMD with 4 components) is normally translated into 4 GEN4 instructions with one for each component. It may result in fewer GEN4 instructions, when some components are not output. To process 16 pixels in parallel using the SIMD8 basic GEN4 instruction set adds another factor of 2. So a 32-line Pixel Shader program may become 256-line long. In order to reduce the burden to the instruction cache, GEN4 architecture employs a simple instruction compression technique available on a subset of instructions. For the Pixel Shader case, one GEN4 instruction in the instruction stream is assigned to process 16 pixels in parallel. When such kind of instruction is encountered, the execution unit automatically creates two instructions before issuing them into the instruction decoder.

For example, a compressed instruction may be in the following form:

*add (**16**) r4.0<8;8,1>:d r2.0<8;8,1>:d r0.0<8;8,1>:d   {**Compr**}*

It will be split by hardware internally into two native GEN4 instructions as:

*add (**8**) r4.0<8;8,1>:d r2.0<8;8,1>:d r0.0<8;8,1>:d*

*add (**8**) r**5**.0<8;8,1>:d r**3**.0<8;8,1>:d r**1**.0<8;8,1>:d   {**SecHalf**}*

The instruction compress technique is expected to be used for Pixel Shader program that runs in channel serial mode for 16 pixels in parallel. It is not clear whether 3D graphics API Vertex and Geometry Shader programs can take advantage of this hardware capability.

The instruction compress technique is also expected to be used for media kernels, where it is quite common to have an operation on a block of data. Instruction compression significantly reduces the burden to the instruction caches. It also allows media kernels to utilize both accumulator registers as implied operands.

11.5.2 Hardware Behavior

Upon encounter an instruction with the Compression Control field set to 10b (Compr), EU hardware treats it as a compressed instruction and converts it into two instructions before sending them down for instruction decode and dispatch. The two decompressed instructions share majority of instruction subfields with the original compressed instruction. The differences are fully described here.

When destination register is an MRF register, a special instruction compression control, ***Compr4***, can be used. ***Compr4*** is the combined condition when instruction compression field is set to **Compr** and the destination register is a MRF register with RegNum[7] set to 1. With ***Compr4***, the second decompressed instruction will have MRF register incremented by 4 registers.

- The first generated instruction
    - Compression Control field is reset to 00b (normal)
    - ExecSize is reduced by half
    - RegNum[7] of the destination MRF register is reset to 0 if the compressed instruction is ***Compr4***.
    - The rests of the instruction word are unchanged
- The second generated instruction
    - Compression Control field is set to 01b (***SecHalf***)
        - If ExecSize of the compressed instruction less than 32, the SecHalf flag determines that the generated instruction (e.g. with ExecSize of 8) uses the 8 MSB of the mask and flag registers. And if a flag register is a conditional destination operand, only the higher 8 bits of a flag register are updated.
        - If ExecSize of the compressed instruction is 32, the whole 16 bits of the mask and flag registers apply to the generated instruction (with ExecSize of 16). And if a flag register is a conditional destination operand, the whole 16 bits of a flag register may be updated. Effective, SecHalf doesn't affect the generation of execution mask in this case.
    - Execution size is reduced by half
    - The breakpoint field is reset (regardless of the value in the original instruction)
    - For a source operand
        - Subject to the following exceptions, for a direct source register, the LSB of the operand's RegNum is set to 1. This effectively moves the register origin by a whole GRF physical register.
            - This rule applies to an accumulator register as an explicit source operand (and it must be acc0 in the compressed instruction). The LSB of the operand's RegNum is set to 1, effectively changing it to acc1. This rule is also consistent with that for an implicit accumulator source.

        - Subject to the following exceptions, for an indirect source register, bit 1 of the address register is set to 1. This effectively moves the address register by one sub-field for the second instruction.
            - The subregister field of the address register must be even aligned, and must be in 1x1 mode.
        - For an implicit accumulator source operand, the ***SecHalf*** flag determines that it is acc1.
            - This rule applies to a compressed instruction regardless of its ExecSize.
        - If a source operand is a scalar (signified by a region with HStride = VStride = 0, also including an immediate operand regardless of it being a scalar or a vector), there is no change.
            - This rule only applies to a direct source but not a register-indirect source. If the intension is to used a single indexed scalar value for a compressed instruction, software must program two-adjacent address subregisters with the same value.
        - If ExecSize of the compressed instruction is 16, the destination has a size of dword and stride of 1 (effectively with a 4-byte stride), and the source operand type is of word size with a horizontal stride of 1, the MSB of the operand's SubRegNum is set to 1. This effectively moves the register origin by half of a GRF physical register.
            - This rule is specifically for mixed data type operation (e.g. Pixel Shader usage).
    - For destination operand
        - For a direct destination GRF register, the LSB of the destination operand's RegNum is ***set*** to 1.
            - This rule applies to an accumulator register as an explicit destination (and it must be acc0 in the compressed instruction). The LSB of the operand's RegNum is set to 1, effectively changing it to acc1. This rule is consistent with that for an implicit accumulator destination.
        - For a destination MRF register (directly addressed only), the destination RegNum is ***incremented*** by 1, if the compressed instruction has ***Compr***.
        - For a destination MRF register (directly addressed only), the destination RegNum[7] is reset to 0 and then RegNum is ***incremented*** by 4, if the compressed instruction has ***Compr4***.
        - For an indirect destination GRF register, bit 1 of the address register is set to 1. This effectively moves the address register by a sub-field for the second instruction.
            - The subregister field of the address register must be even aligned, and must be in 1x1 mode.
        - For an implicit accumulator destination operand, the ***SecHalf*** flag determines that it is acc1.

As all 16 entries of the mask stack are pushed or popped together each time a mask stack is updated and the mask stack is implemented by counters, it is essential to not mix the execution channel select with the use of ***SecHalf*** when nesting branch/loop instruction blocks. In other words, mask values that are pushed onto a stack must all have been generated from instructions using the same Compression Control value (such as normal, ***SecHalf*** and ***Compr***).

The Thread Control field is unchanged for both generated (decompressed) instructions. If this field is set to **Switch** in the compressed instruction, both decompressed instructions will cause forced thread switch.

Compr4 is not supported.

[DevBW, DevCL] Errata: If ExecSize is 16, the implicit accumulator is forced to acc0, even if ***SecHalf*** compression control is set. Normally, ***SecHalf*** determines that acc1 is the implicit accumulator. However, this is overruled by ExecSize of 16. This restriction implies that instructions with implicit accumulator as a source (such as mac, mach) cannot be used for a compressed instruction with ExecSize of 32.

[DevBW, DevCL] Errata: When ***SecHalf*** compression control is used, even with ExecSize of 16, only the second half of the flag register may be used (for example as conditional destination or as prediction source).

11.5.3 Rules and Restrictions

In order to reduce the hardware complexity, the following rules and restrictions apply to the compressed instruction:

- Instruction compression is on a per instruction basis. Compressed instruction and normal instruction can be intermixed in any program.
- A compressed instruction must be in Align1 access mode. Align16 mode instructions cannot be compressed.
- A compressed instruction with indirect addressed operands for any operand (source, destination) is allowed.
  - *Rational*: Commonly used in media kernels.
- Operand Alignment Rule: With the exceptions listed below, a source/destination operand in general should be aligned to **even** 256-bit physical register with a region size equal to two 256-bit physical registers.
  - *Rational*: This allows an insertion of 1 to the LSB of the RegNum to create the second instruction.
- Exception A to the Operand Alignment Rule: for compressed instructions operating on packed words (destination is not dword size with stride of one), an operand with a non-zero subregister field is also allowed, as long as its register number is an **even** number.
  - *Rational*: This allows an insertion of 1 to the LSB of the RegNum field to create the second instruction. And hardware does not modify the subregister field.
- Exception B to the Operand Alignment Rule: a source operand with scalar replication (with both HStride = 0 and VStride = 0) is allowed to have address unaligned to 256-bit.
  - *Rational*: There is no change to this source operand to create the second instruction.

- Exception C to the Operand Alignment Rule: For a compressed instruction with dword destination with stride of one, for a source operand with packed word data elements (type = W or UW, and HStride = 1), the register origin must be aligned to 256-bit physical register (i.e. SubRegNum = 0).
  - If a source type is B or UB, HStride must be either 0 (scalar) or 2 (packed word).
- Exception D to the Operand Alignment Rule: a destination operand to MRF must be aligned to 256-bit physical register (either **even** or **odd**).
  - *Rational*: This allows more efficient use of the rare MRF resource, considering the fact that many messages have a single register message header field.
- *ExecSize* must be 16 or 32
  - *Rational*: Allows hardware to decrement by 1 to the encoded *ExecSize* field.
  - *Note*: Most commonly used mode for Pixel Shader is that *ExecSize* = 16. Most commonly used mode for media kernels is that ExecSize = 32 with packed word operations. There are also some media usage of compressed instruction with ExecSize = 16.
- Instruction compression does not apply to branch instructions such as jmpi, send, do, while, wait, etc.
- A compression instruction in general cannot address (read and/or write) ARF registers.
  - Exception to this rule is the explicit or implicit access to the accumulator registers and implicit access to flag registers and mask registers, as well as indirect addressed GRF registers.

11.5.4 Usage Examples

Some examples are provided in this section to help visualizing the effect of compressed instruction on source/destination operands. It is for illustration purpose only; it is not intended as comprehensive usage coverage.

Figure 11-19 shows examples of directly-addressed vector operands in a compressed instruction. In these cases, hardware inserts one to the LSB of the RegNum field of a GRF operand (or increment one to RegNum of a MRF operand) for the second decompressed instruction.

- In Figure 11-19(a), the operand occupies two adjacent rows of register space with the first decompressed instruction operating on the first eight elements and the second decompressed instruction on the second eight elements. It may be a dword source or destination operand of a SIMD16 compressed instruction. The operand register address must be even aligned to a GRF register or register aligned to a MRF register.
- Figure 11-19(b) shows a result when the register address of the dword (source) operand of a SIMD16 compressed instruction is aligned to odd GRF register. As inserting one to RegNum field doesn't change a thing, the same eight elements are repeated for the two decompressed instructions.
- Figure 11-19(c) shows a word/byte source/destination operand in a SIMD16 compressed instruction. RegNum field is an even number; however, SubRegNum here is not zero. So the elements of a decompressed instruction only occupy a portion of a GRF register.

Figure 11-19. Direct addressed vector operands in a compressed instruction

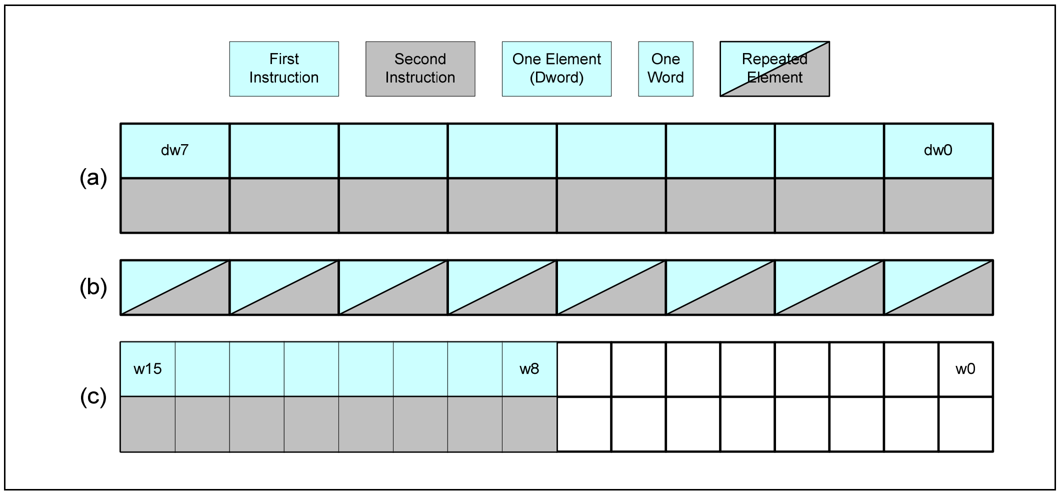


Figure 11-20 shows a case of a packed-word vector operand in a dword SIMD16 compressed instruction. When the destination stride is 4-byte, instead of inserting one to the LSB of RegNum field of src0, hardware will insert one to the MSB of SubRegNum field, as src0 is a packed-word vector (horizontal stride equals to 2-byte). This is specifically designed for Pixel Shader application.

Figure 11-20. A packed-word operand in a dword SIMD16 compressed instruction

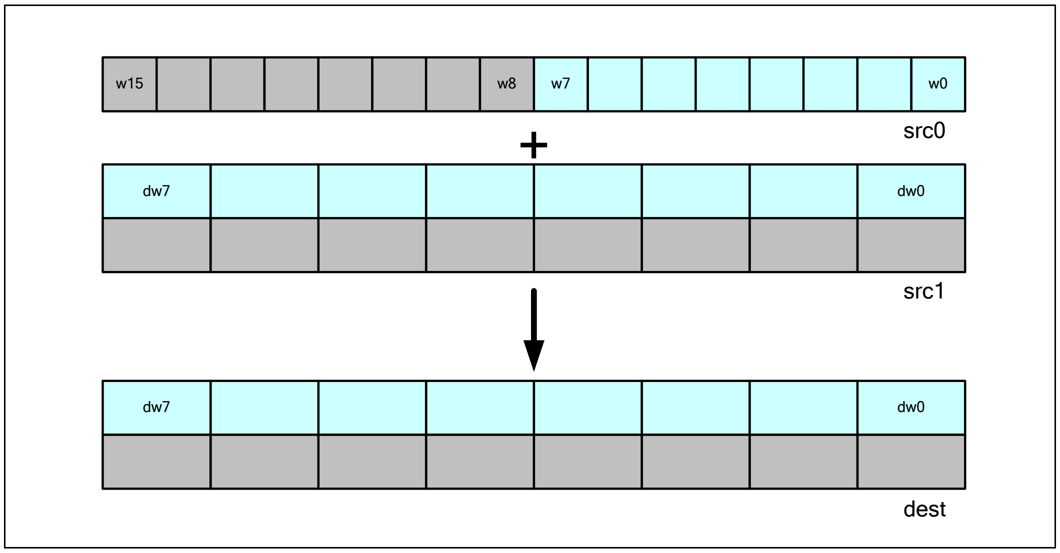


Figure 11-21 shows cases of scalar sources in a compressed instruction. There is no alignment restriction for a scalar source in a compressed instruction,

Figure 11-21. Dword and word (or byte) scalar source in a compressed instruction

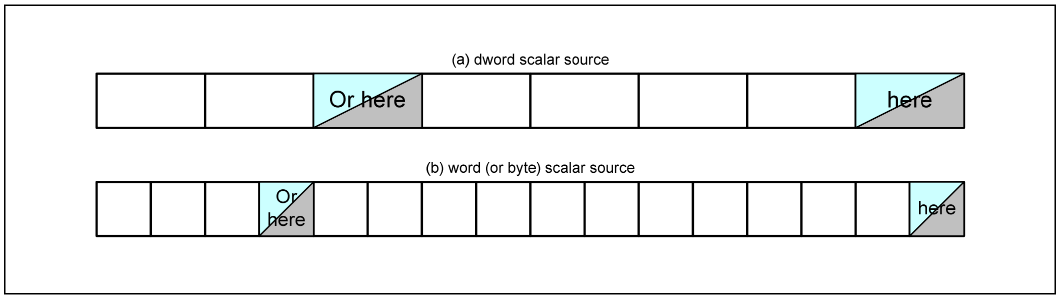


Figure 11-22 shows cases of indirectly-addressed operands in a compressed instruction. Hardware inserts one to bit 1 of SubRegNum field of the address register for the second decompressed instruction. This capability may be used for register regions that are not supported by the previously-mentioned cases. For example, source0 of the following eight instructions may be compressed with indirect-addressed.

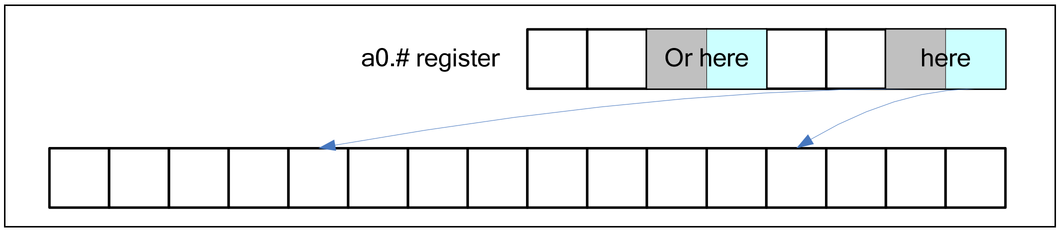
Direct-addressed instructions:

```
add.sat (8) r26.0<2>:ub  r2.0<8;8,1>:w   r26.0<16;8,2>:ub {Align1 }
add.sat (8) r27.0<2>:ub  r2.8<8;8,1>:w   r27.0<16;8,2>:ub { Align1 }
add.sat (8) r28.0<2>:ub  r3.0<8;8,1>:w   r28.0<16;8,2>:ub { Align1 }
add.sat (8) r29.0<2>:ub  r3.8<8;8,1>:w   r29.0<16;8,2>:ub { Align1 }
add.sat (8) r30.0<2>:ub  r4.0<8;8,1>:w   r30.0<16;8,2>:ub { Align1 }
add.sat (8) r31.0<2>:ub  r4.8<8;8,1>:w   r31.0<16;8,2>:ub { Align1 }
add.sat (8) r32.0<2>:ub  r5.0<8;8,1>:w   r32.0<16;8,2>:ub { Align1 }
add.sat (8) r33.0<2>:ub  r5.8<8;8,1>:w   r33.0<16;8,2>:ub { Align1 }
```

Compressed instructions with indirectly-addressing mode:

```
mov (1) a0.1<1>:d   0x00500040:d
add.sat (16) r26.0<2>:ub  r[a0.2,0]<8;8,1>:w   r26.0<16;8,2>:ub { Align1, Compr }
add.sat (16) r28.0<2>:ub  r[a0.2,32]<8;8,1>:w   r28.0<16;8,2>:ub { Align1, Compr }
add.sat (16) r30.0<2>:ub  r[a0.2,64]<8;8,1>:w   r30.0<16;8,2>:ub { Align1, Compr }
add.sat (16) r32.0<2>:ub  r[a0.2,96]<8;8,1>:w   r32.0<16;8,2>:ub { Align1, Compr }
```

Figure 11-22. Indirect-addressed source / destination operand in a compressed instruction



11.6 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.

11.7 Creating Conditional Flags

FPU will output 2 sets of conditional signals, 1 set will be generated from before the adder outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. 1 set will be generated from the final results after clamping and re-normalizing/format conversion logic, and we will call this the post conditional signals. The post conditional signals are used for fusing the compare instruction. *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 will be used to generated flags for CMP/CMPN instructions only, this logically does the compare of the 2 input sources. The post conditional signals will be used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

CMPN with both sources are NaN is a don't care case since 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 directly from the adders, without going through any kind clamping, normalizing, or format conversion logic.
- **pre_zero** bit: this bit reflects whether the value of the adder results are zero, again this should be obtained before any kind 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 value of the adder results are zero after all the clamping, normalizing, or format conversion logic.
- **OF** bit: this bit reflects whether an overflow occured in any of the compution of the current instruction, including clamping, re-normalizing, and format conversion.
- **INC** bit: The increment bit is only used for RNDU, RNDE, and RNDZ instructions to convey the information whether an additional increment is needed for the execution channel for the given round instruction. It always returns 0 for RNDD and is undefined for other instructions.
- **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 set to 0.
- **NS0** bit: The NaN bit indicates whether source 0 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 set to 0.
- **NS1** bit: The NaN bit indicates whether source 1 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 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.

Flag Generation for CMP instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, and **.le**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
'**.e**'	Equal-to	**(pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
'**.ne**'	Not-Equal-to	**!(pre_zero & !(NS0 | NS1)).** This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier '**.e**'.
'**.g**'	Greater-than	**(!pre_sign & !pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.ge**'	Greater-than-or-equal-to	**((!pre_sign | pre_zero) & !(NS0 | NS1)).** This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.l**'	Less-than	**(pre_sign & !pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.le**'	Less-than-or-equal-to	**((pre_sign | pre_zero) & !(NS0 | NS1)).** This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.

Flag Generation for CMPN instructions (The supported Conditional Modifiers are **ge**, and **.l**)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
'**.ge**'	Greater-than-or-equal-to	**(!pre_sign | pre_zero | (NS1 & (Opcode==CMPN | OPcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)).** This conditional modifier tests whether source0 is greater than or equal to source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.
'**.l**'	Less-than	**((pre_sign & !pre_zero) | (NS1 & (Opcode==CMPN | Opcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)).** This conditional modifier tests whether source0 is less than source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.

Flag Generation for All Arithmetic Instructions other than CMP/CMPN instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, **.le**, **.r**, **.o**, and **.u**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
'**.e**'	Equal-to	**(post_zero & !NC).** This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
'**.ne**'	Not-Equal-to	**!(post_zero & !NC).** This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier '**.e**'.
'**.g**'	Greater-than	**(!post_sign & !post_zero & !NC).** This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.ge**'	Greater-than-or-equal-to	**((!post_sign | post_zero) & !NC).** This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.l**'	Less-than	**(post_sign & !post_zero & !NC).** This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.le**'	Less-than-or-equal-to	**((post_sign | post_zero) & !NC).** This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.r**'	Round-Increment	**(IN & (!OF)).** This conditional modifier tests whether the rounding result (of a RNDx instruction) requires increment. Normally, the condition is true if IN is true. However, if overflow occurs for the execution channel (OF is true), the condition is force to false.
'**.o**'	Overflow	**(OF).** This conditional modifier tests whether the computed result causes overflow – the computed result is outside the range of the destination data type. All other internal conditional signals are ignored.
'**.u**'	Unordered	**(NC).** This conditional modifier tests whether the computed result is a NaN (unordered). All other internal conditional signals are ignored.

Programming Notes:

CMPN should be used ONLY to emulate MIN/MAX operations, and only the following macros should be used for MIN and MAX operations.

<u>Macro for MIN:</u>

```
CMPN.l.f0.0   null   s0  s1

(f0.0) SEL    dst s0  s1
```

<u>Macro for MAX:</u>

```
CMPN.ge.f0.0 null   s0  s1

(f0.0) SEL    dst   s0  s1
```

11.8 Destination Hazard

GEN4 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 GEN4 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, GEN4 hardware automatically takes care of the destination hazard by not issuing the second instruction until the destination scoreboard is cleared. However, for certain cases, in fact for most cases, such destination hazard indicated by the register scoreboard is false, causing unnecessary delay of instruction issuing. This may result in lower performance. The destination dependency control field in the instruction word {*NoDDClr*, *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 100% certainty that it is a false destination hazard, programmer should reply on hardware to result the dependency.

As the destination dependency control field does not apply to *send* instruction, there is only one condition that a programmer may use the {*NoDDClr*, *NoDDChk*} capability.

- If none of the two instructions is *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.

11.9 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>.

11.10 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).

12 Exceptions

12.1 Introduction

The Gen4 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, was well as illegal conditions and debug features.

The following exception-types are supported:

Type	Trigger / Source	Sync/Async Recognition
MaskStack Overflow / Underflow	Hardware	Synchronous (w/ special case for 'do'; see 12.4.3)
Software Exception	Thread code	Synchronous
Breakpoint	A bit in the instruction word Triggering a debug snapshot	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver, and delivered to an EU as part of a new-thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's CR0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception-type recognition across hardware threads. The exception enables in the CR0.1 register are r/w, and thus can be enabled/disabled via software at anytime during thread execution.

The exception handling mechanism relies on the "system routine", a single subroutine which 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 32b 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.

12.2 Exception-Related Architectural Registers

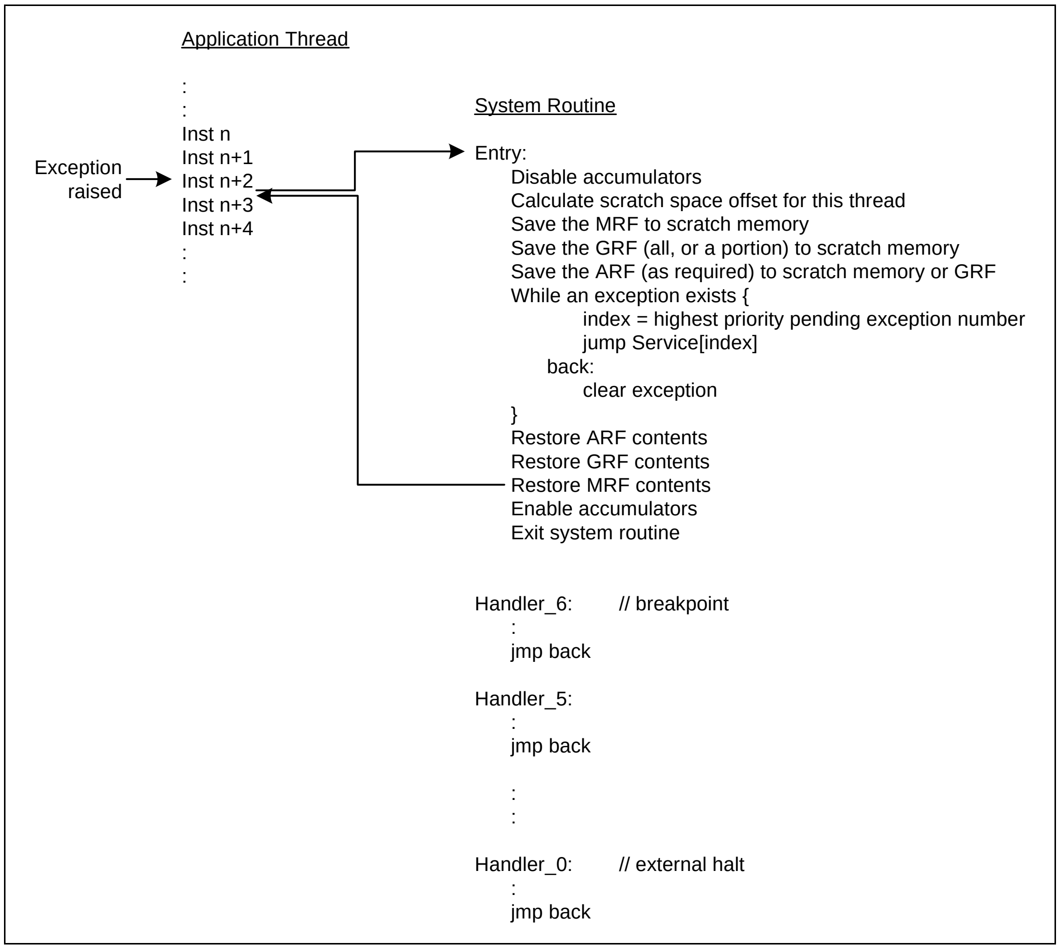
Exception-related registers are defined in architectural register CR0.0 through CR0.2. These registers are instantiated on a per-thread basis providing each hardware thread with unique control over exception recognition and handling. The registers provide the capability to mask exception types, determine the type of raised exception, provide storage 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 an atomic R-M-W operation when accessing these registers.

12.3 System Routine

12.3.1 General Flow of the System Routine

The following diagram illustrates the basic flow of exception handling and structure of the system routine.



```
Entry:
    Disable accumulators
    Calculate scratch space offset for this thread
    Save the MRF to scratch memory
    Save the GRF (all, or a portion) to scratch memory
    Save the ARF (as required) to scratch memory or GRF
    While an exception exists {
            index = highest priority pending exception number
            jump Service[index]
        back:
            clear exception
    }
    Restore ARF contents
    Restore GRF contents
    Restore MRF contents
    Enable accumulators
    Exit system routine


Handler_6:      // breakpoint
    :
    jmp back

Handler_5:
    :
    jmp back

    :
    :

Handler_0:      // external halt
    :
    jmp back
```

12.3.2 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 will be issued until the system routine has executed and returned control back to the application thread.

After a exception is recognized by hardware, the EU saves the thread's IP into its AIP register (CR0.2), an then moves the system routine offset, SIP, into the thread's IP register. At this point the next instruction to issue for that thread will be the first instruction of the system routine.

The system routine maintains the same execution priority, GRF and MRF register space, and thread state as that of the application thread from which it was invoked. Due to the assuming the same priority, there may be significant absolute time between exception being raised and the actual invocation of 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 to issue 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 such time that the register retires.

Exception processing is non-nested within an 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 will be 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 prior to using any instruction which modifies an accumulator, or manually save/restore the accumulators (to GRF registers or system thread scratch memory) around the system routine. Saving/restoring accumulators, including their extended precision bits, can be accomplished by a short series of mov's and shifts of the accumulator register. Also note 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 architectural registers across the system routine. The system routine must ensure that these values are preserved (see Section 12.3.7 for related discussion).

12.3.3 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 CR1. Failure to do so will force the thread's execution to re-enter the system routine prior to any further instructions being executed from that application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset prior to exit.)

The system routine may choose to loop under 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 of CR0.0. Upon clearing this bit, the value of the AIP architectural register (CR0.2) is restored to the thread's IP register and, with no further exceptions pending, execution resumes that address. The system routine must follow any write to 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).

12.3.4 System-IP (SIP)

The System IP (SIP) is a 16B-aligned 32b offset of the first instruction of the system routine, relative to the General State Base Address. It is set via the STATE_IP command to the command streamer. The upper 28b of the 32b address is automatically delivered to all Gen4 EUs.

When the system routine is invoked, the application thread's current IP is first saved into the AIP field of the thread's architectural register 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 at the first instruction of the system routine. Upon system routine completion, the value held in AIP is returned to IP and execution continues on the application thread at the place where the exception was recognized.

12.3.5 System Routine Register Space

The system routine uses the same GRF and MRF space at the thread which invoked 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 to all threads. It should also be noted that the system routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

Only one 32b GRF location, R0.4, is reserved for system routine usage. This is sufficient to allow the system routine to calculate the appropriate offset of its private scratch memory in the larger system-scratch memory space (as dictated by binding table entry 254). The offset is left as a driver convention, but likely based on a combination of Thread and EU IDs (see example system handler in section 12.3.6).

Other than the reserved R0.4 register field, there is no explicit GRF register space dedicated to the system routine, and any GRF needs must be accomplished via: (a) convention between the system routine an application thread, or (b) the system routine temporarily spilling the thread's GRF register contents to scratch memory, and restoration prior to system routine exit.

No persistent storage is natively allocated to the system routine, although a driver implementation may choose to carve out a piece of system scratch memory though it own convention.

Any parameter passing to the system routine (for use by s/w exceptions) is performed via the GRF based on a system-thread/application-thread convention.

12.3.6 System-Scratch Memory Space

There is a single unified system-scratch memory space per context shared by all EUs. It is anticipated that block is further partitioned into a unique scratch sub-space per-thread via convention implemented in the system routine, with a each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread is based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architectural register SR0.0.

```
    Per_Thread_Block_Size = System_Scratch_Block_Size / (EU_Count * 
Thread_Per_EU);

    Offset = (SR0.0.EID * Threads_Per_EU + SR0.0.TID) * Per_Thread_Block_Size;

                    where in Gen4...
                        Threads_Per_EU = 4
                        EU_Count = 8
                        System_Scratch_Block_Size is a driver choice
```

Access to the system-scratch memory is performed through the Data Port via linear single-register or block-based read/write messages. The driver may choose to use any binding table index for system-scratch surface description. As a practical matter, the same index is expected to be used across all binding tables, as the index is typically hard coded in dataport messages used within the system routine coupled with the fact that a single system instance routine is used for all threads. Read/write messages to the Data Port contain the address of the binding table (provided in R0 of all threads) and an offset, from which the Data Port calculates the final target address.

The size of the overall system-scratch memory block is a function of the system routine's feature set, but for debug purposes should be, at a minimum, of size sufficient to cover storing the entire GRF, MRF, flags, and architectural register set for all hardware threads. Additional per-thread memory may also be required for mask-stack spill/fill, system routine constants, and any global persistent storage needs of the system routine.

It is expected that the system-memory block is allocated by the driver at context-create time and remains persistent at a constant memory address throughout the context's lifetime.

12.3.7 Conditional Instructions Within System Routines

It is expected that most, if not all, control flow with in 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 simply to implement the save/restore as part of the entry/exit code sequence, using an available GRF register-pair as storage location. Once saved, the stacks should be reset to their empty condition, namely depth = 0 and top-of-stack = 0xFFFFFFFF.

12.3.8 Messages in System Routines

The system routine uses the same MRF space as the thread on whose behalf the system routine was invoked. To allow the thread to resume with the same state as prior to the system routine invocation, the thread's MRF contents must be preserved across a system routine invocation. If the system routine requires MRF space for messages, it must manually save and restore the MRF locations which it uses.

Note that the MRF can only be used as an instruction's destination register, not a source. Therefore there is no option to save the MRF to the GRF. Thus the system routine should save the MRF contents to its dedicated scratch space. By convention it is recommended that MRF register m0 be reserved for system-thread use. This allows the system routine enough space to construct an initial Data Port write message starting at m0 without corrupting any MRF registers, facilitating a complete save/restore of the MRF by the system-thread.

12.3.9 Use of 'NoDDClr'

The Gen4 instruction word defines an instruction option 'NoDDClr' which 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 the 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 w/o dependency checks, and the last instruction 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 should an exception be raised within a 'NoDDClr' code block. In this case, there exists the potential for the system routine to hang while attempting to save/restore the code blocks destination register, as the outstanding dependency on that register will not clear until the final instruction of the block is executed – sometime after the system thread returns. Should the system routine attempt to use that register, the system routine will hang waiting on a dependency to clear from an instruction which has not yet been issued.

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 minimize the number of cases of such, 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 which may generate exceptions should not be placed within 'NoDDClr' blocks. This includes most conditional branch instructions (if, do, while, ...) as well as breakpoints explicitly in the instruction stream or triggered by the debug facilities in the IL1 instruction cache.
- To guarantee that r0 is visible in system scratch memory for debug purposes under any condition, (a) threads should not use 'NoDDClr' on r0, and (b) the system routine should move and send r0 to system scratch memory as a single register (as opposed to a block of 'n' registers). For practical reasons related to dataport message sizes, it may be beneficial to extend this restriction to r1 as well.
- 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 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.

12.4 Exception Descriptions

12.4.1 'Illegal' opcode

The Gen4 ISA defines a single 'illegal' opcode. The byte value of the 'illegal' opcode is selected to be 0x00 due to it being a likely byte-value encountered by a wayward instruction pointer value. The 'illegal' instruction raises an exception prior to issue and operates as a 'nop' when issued down the execution pipeline. (Specifically, the opcode acts a 'nop', although other non-opcode instruction attributes still apply).

12.4.2 Undefined opcode

All undefined opcodes in the 8b opcode space are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing it 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 section 12.4.1. Note that the memory location of the offending opcode remains modified and may be queried if desired to determine its original value.

12.4.3 MaskStack Overflow / Underflow

Hardware-based mask-stacks are used for saving and restoring mask values for nested ifs and loops. These structure are limited to 16 levels of nesting. Thus, in the face of normal execution where nesting levels exceed 16, the hardware mask-stacks may occasionally overflow.

Upon reaching maximum capacity, the hardware sets the appropriate LStack/IStack Overflow Exception Status and Control bits provided in the thread's architectural register CR0.1. This exception may be used by the system routine to extend the depths of mask-stacks by saving the stack contents and current depth to scratch memory prior to overflow. After saving, the stacks are reset and execution continues for another multiple of 16 nesting levels where a further exceptions are generated and the mask-stacks saved.

Likewise, as a thread exits if or loop nesting levels, masks values are restored automatically from the hardware mask-stack via 'pop' operations. A stack underflow may occur if the nesting level crosses a multiple of the hardware capacity. The LStack/IStack Underflow exception if provided for restoring stack values from where they had been previously saved.

As an alternative implementation, the driver may choose to manage the mask-stacks manually, by maintaining nesting level counts during the thread's JIT process, and inserting code snippets to save/restore the stacks prior to overflow/underflow.

12.4.4 Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered through of 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 AIP instruction pointer is incremented prior to system-routine entry, therefore causing execution to resume at the subsequent application-thread instruction when the system routine is exited.

12.4.5 Breakpoint

A bit ('DebugCtrl') is defined in the instruction word which, when set, causes as breakpoint exception to be triggered prior to the associated instruction being executed. This bit can be set statically in the instruction word or may be dynamically set by hardware upon triggering a snapshot event in the Instruction L1 ("Instruction L1 Cache Breakpoint Address" registers in the Debug chapter). Note that the dynamic setting mechanism of this bit is on an instruction-fetch by instruction-fetch basis (i.e. is non-sticky in the L1). The system routine is expected to handle this exception by providing debug facilities, such as dumping/restoring the thread's state to programmer-visible memory for inspection.

Upon return from the system routine, the instruction which contained the breakpoint is re-issued. To prevent a subsequent breakpoint exception from being generated by the same instruction, breakpoint recognition may be suppressed for one instruction via the Breakpoint Suppress field of CR0.0. This field is typically set by the system routine prior to exiting a breakpoint exception.

For normal breakpoint operations, the system routine must clear the "Breakpoint Exception Status and Control" field of CR0.1 prior to returning from the system routine.

A single-stepping capability may be implemented by leaving the "Breakpoint Exception Status and Control" set, and clearing the Breakpoint Suppress field prior to system routine exit. This combination causes the instruction associated with the breakpoint to be reissued, this time with the breakpoint suppressed, and then re-entry to the system routine prior to the subsequent instruction due to the lingering breakpoint exception that remained un-cleared.

12.4.6 External Halt

A 'halt' exception may occur upon satisfying a debug snapshot trigger or via direction manipulation of a MMIO bit by driver software (see the External Halt on Snapshot bit, or Force External Halt bit of the TD-Debug Control register - Debug chapter). The halt exception is sent to all EUs simultaneously (although no guarantee is made as to recognition in identical clocks). An EU recognizes this condition internally by generating an External Halt exception. A likely implementation of a handling routine would dump the thread's state to programmer-visible memory (such as the system routine's scratch space) for inspection and debugging purposes. Although generally recognized within a few clocks, there is no specification as to the latency between triggering the Halt condition and it being recognized by an EU.

12.5 Events Which Do Not Generate Exceptions

The following conditions are either not recognized or do not generate an exception.

Illegal Instruction Format

This includes malformed instructions in which the opcode is legal, but the source or destination operands, or instruction attributes are not compliant 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 Gen4 does not support self-modifying code, therefore the driver (perhaps in some form of debug mode) has an opportunity to detect such cases before the thread is placed in service.

Malformed Message

A messages contents, destination registers, lengths, and descriptors are not interpreted in anyway by the execution units. Errors in specifying any of these fields do not raise exceptions in the execution unit but may be detected and reported by the shared functions (see the Debug chapter for details).

GRF Register Out-of-Bounds

Unique GRF storage is allocated to each thread which, at a minimum, satisfies that 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 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 threads register space, thus there is no possibility of direct register manipulation from an out-of-bounds register access.

MRF Register Out-of-Bounds

A fixed amount of MRF register space is allocated for each thread, namely m0 through m15. References to MRF registers beyond m15 do not generate exceptions. Depending on implementation details, out-of-bounds register numbers may alias to in-bounds register numbers, although this functionality should not be relied upon by the thread.

Hung Thread

There is no hardware mechanism in the execution units to detect a hung thread, and should it occur, the thread remains hung indefinitely. It is the expectation 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 Gen4 EUs implement a full 32b instruction address range (with the 4 lsb's don't care), making it possible for a thread to attempt to jump to any 16B aligned offset in the 32b address space. The EU itself does not provide any type of address checking on its instruction request stream sent to the memory/cache hierarchy, although various memory address related error conditions are reported through the Memory Interface Registers (specifically "Page Table Error Register").

FPU Math Errors

The EU's floating point units have defined behavior for traditional floating point errors and do not generate exceptions. Therefore there is no support for signaling FPU math errors as exceptions.

Destination Register Overflow

Depending on source operand contents, destination register size, and operation being performed, overflows may occur in the EU's pipeline. These are not flagged as exceptions and software must explicitly check the overflow bit in the thread's architectural register if overflow is a concern.

12.6 System Handler Example

The following code sequence illustrates some concepts of the system routine. It is intended to be just a shell, without getting into the specifics of each exception handler. The example frees enough MRF and GRF space to get the routine started, then jumps to the handler for the specific exception. Many other implementations are also valid, including single exception servicing (as opposed to looping) per invocation, and saving only the GRF or MRF space required by the exception being serviced.

```
         #define ACC_DISABLE_MASK                0xFFFFFFFD
         #define MASTER_EXCP_MASK                0x7FFFFFFF
         #define SYSROUTINE_SCRATCH_BLKSIZE      16384     //for example

         // --- SharedFunc IDs ---
         #define DPR                             0x04000000
         #define DPW                             0x05000000

         // --- message lengths ---
         #define ML5                             0x00500000
         #define ML9                             0x00900000

         // --- response lengths ---
         #define RL0                             0x00000000
         #define RL4                             0x00040000
         #define RL8                             0x00080000

         // --- dataport block sizes ---
         #define BS1_LOW                         0x0000
         #define BS1_HIGH                        0x0100
         #define BS2                             0x0200
         #define BS4                             0x0300

         // --- Scratch Layout ---
         #define SCR_OFFSET_MRF                  0
         #define SCR_OFFSET_GRF                  512       // + 16 reg
         #define SCR_OFFSET_ARF                  512 + 4096    // + 16 + 128
reg



         // --- Write Dataport constants ---
         // target=dcache, type= oword_block_wr, binding_tbl_offset=0
         #define DPW                             0x000

         // --- Read Dataport constants ---
         // target=dcache, type= oword_block_rd, binding_tbl_offset=0
         #define DPR                             0x000

Sys_Entry:
         // --- disable accumulator for debug routine ---
```

```
          and   (1) cr0.0 cr0.0 ACC_DISABLE_MASK     {NoMask}

          // --- calc scratch offset for this thread into r0.4 ---
          shr   (1) r0.4 sr0.0:uw 6                      {NoMask}
          add   (1) r0.4 r0.4 sr0.0:ub                   {NoMask}
          mul   (1) r0.4 r0.4 SYSROUTINE_SCRATCH_BLKSIZE     {NoMask}

          // --- setup m0 w/ block offset
          mov   (8) m0 r0                                {NoMask}

          // --- save mrf 7...0; (may choose to save the whole mrf)
          add   (1) m0.2 r0.4 SCR_OFFSET_MRF             {NoMask}
          send  (8) null m0 null DPW|ML9|RL0             {NoMask}

          // --- save mrf 8...15; (optional; req'ed if sys-routine stays w/in
mrf7-0)
          mov   (8) m7 r0                                {NoMask}
          add   (1) m7.2 r0.4 (SCR_OFFSET_MRF + 256) {NoMask}
          send  (8) null m7 null DPW|ML9|RL0             {NoMask}

          // --- save r0..r1 to system scratch ---
          // --- (Note: done as a single register to guarantee external
          // ---  visibility – see "Use of 'NoDDClr'" in Excpetions PRM chapter
          mov  (16)  m1 r0                               {NoMask}
          send (8)  m0 null null DPW|ML2|RL0             {NoMask}

          // --- save r2..r3 to free some room
          mov  (16) m3 r2                                {NoMask}
          add  (1)  m0.2 r0.4 SCR_OFFSET_GRF + 64        {NoMask}
          send (8)  m0 null null DPW|ML4|RL0             {NoMask}

          // --- save r4..r7 to free some room (optional, depending on needs)
          mov  (16) m8 r4                                {NoMask}
          mov  (16) m10 r6                               {NoMask}
          add  (1)  m7.2 r0.4 (SCR_OFFSET_GRF + 128) {NoMask}
          send (8)  m7 null null DPW|ML5|RL0             {NoMask}

          // --- save r8..r11 to free some room (optional, depending on needs)
          mov  (16) m1 r8                                {NoMask}
          mov  (16) m3 r10                               {NoMask}
          add  (1)  m0.2 r0.4 (SCR_OFFSET_GRF + 256) {NoMask}
          send (8)  m0 null null DPW|ML5|RL0             {NoMask}

          // --- save r12..r15 to free some room   (optional, depending on
needs)
          mov  (16) m8 r12                               {NoMask}
          mov  (16) m10 r14                              {NoMask}
          add  (1)  m7.2 r0.4 (SCR_OFFSET_GRF + 384) {NoMask}
          send (8)  m7 null null DPW|ML5|RL0             {NoMask}

          // --- save ARF registers (optional, depending on use) ---
          // flags, maskstacks, others...

          // --- save f0.0 ---
           mov (1)  r1.0:uw f0.0                         {NoMask}

Next:     // --- exceptions pending? If not, exit ---
          cmp.e (1) f0.0 cr0.4:uw 0:uw                   {NoMask}
          (f0.0) mov (1) IP EXIT                         {NoMask}

          // --- find highest priority exception ---
          lzd  (1) r1.1:uw cr0.4:uw                      {NoMask}

          // --- jumptable to service routine ---
          jmpi (1)  r1.1:uw                              {NoMask}
          mov  (1)  IP CRService_0                       {NoMask}
          mov  (1)  IP CRService_1                       {NoMask}
          mov  (1)  IP CRService_2                       {NoMask}
          // :
          // :
```

```
          // :
          mov  (1)  IP CRService_15                      {NoMask}

          mov  (1)  IP Next
Service_0:
          // clear exception from CR0.1
          // perform service routine
          // jump to exit (or if looping on exceptions, go to next loop)

          // :
          // :

Service_15:
          // clear exception from CR0.1
          // perform service routine
          // jump to exit (or if looping on exceptions, go to next loop)

Exit:
          // --- restore f0.0 ---

          // --- restore ARF registers (as required) ---
          // flags, maskstacks, others...

          // --- restore r12..r15 ---
          // --- restore r8..r11 ---
          // --- restore r4..r7 ---
          // --- restore r0..r3 ---

          // --- restore m8..m15 ---
          // --- restore m0..m7 ---

          // --- clear Master Exception State bit in CR0.0
          and  (1)  cr0.0 cr0.0 MASTER_EXCP_MASK
          nop  (16)
```

Below is a code sequence to programmatically clear the GRF scoreboard in the case of a timeout waiting on a register that may never return:

```
           // At this point, all we know is we have a hung thread. We'd like to
copy the
       // GRF to scratch memory to make it visible for debug purposes, but there
may be
       // a register that is hung w. an outstanding dependency. To get around
       // any hung dependency, we can walk the GRF using NoDDChk, using execution
mask
       // of f0 = 0 so we don't touch the register contents.

Clear_Dep:
            mov f0  0x00
       (f0) mov r0 0x00 {NoDDChk}
       (f0) mov r1 0x00 {NoDDChk}
       (f0) mov r2 0x00 {NoDDChk}
           ...
           ...
       (f0) mov r127 0x00 {NoDDChk}

       // GRF scoreboard now cleared.
```

§§

13 Instruction Set Summary

13.1 Instruction Set Characteristics

13.1.1 SIMD Instructions and SIMD Width

GEN4 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 GEN4 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 GEN4 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 GEN4 instruction works on twice as many SIMD data as that for a non-compressed GEN4 instruction. Non-compressed instructions are also referred to as 'native' instructions. A compressed instruction is converted into two native instructions by the instruction dispatcher in the EU.

GEN4 instructions are executed on a narrower SIMD execution pipeline. Therefore, GEN4 native instructions take multiple execution cycles to complete. See Table 13-1 for parameters for difference device hardware.

Table 13-1. GRF instruction execution parameters in device hardware

Device	Execution Pipeline	Native Instruction		Compressed Instruction	
	Width (bits)	Max Width (bits)	Min Execution Cycles (clocks)	Width (bits)	Min Execution Cycles (clocks)
[DevBW]	128	256	2	512	4
[DevCL]	128	256	2	512	4

13.1.2 Instruction Operands and Register Regions

Majority of GEN4 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***, ***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 that contains 8 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.

13.1.3 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.

Format conversion of the result is done through destination format specifier; normally, there is no need for specific format conversion instructions.

Mixed format operation, where the two source operands are in different formats, is supported through source type specifier.

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.

13.2 Instruction Machine Formats

This section shows the machine formats of the GEN4 instruction set. The instructions in GEN4 architecture have fixed length of 128 bits. Out of the 128 bits, there are 111 bits in use, and the remaining bits are reserved for future extensions. One instruction consists of instruction fields that control various stages of execution of the instruction. These fields are roughly groups into the 4 doublewords as the following.

- Instruction Operation Doubleword (DW0) contains the opcode and other general instruction control fields.
- Instruction Destination Doubleword (DW1) contains the destination operand (<dst>) and the register file and type of source operands.
- Instruction Source-0 Doubleword (DW2) contains the first source operand (<src0>) and flag register number
- Instruction Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>).

Table 13-2 depicts the details of the organization of fields in the 128-bit instruction word based on the Addressing Mode and Access Mode of an instruction. Definitions for individual instruction fields are provided in the following sections.

The ***send*** instruction is also shown in the talbe as it has some unique instruction fields. For example, the message descriptor (plus EOT) occupies the whole DW3, and the immediate destination register overlaps with the Conditional Modifier field. The rest of fields in DW0-3 follows the definition on the left, depending on Addressing Mode and Access Mode of the ***send*** instruction.

Not shown is for immediate operands. When an immediate source is present in an instruction, it always occupies the whole DW3 with a 32-bit value.

Support for indirect addressing for <src1>, as shown by the gray areas in Table 4-2 is device dependent. See Table 11-30 (Indirect source addressing support available in device hardware) in ISA Execution Environment for details.

Table 13-2. GEN4 Instruction Format

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	AddrMode = Direct		AddrMode = Indirect		SEND	
					AccessMode = Align16	AccessMode = Align1	AccessMode = Align16	AccessMode = Align1	MsgDesc Imm	MsgDesc Reg
	1	127	127	**0**					*EOT*	
	6	126	121	**0**						
	4	120	117	4	*Src1.VertStride*		***Src1.VertStride***			
	1	116	116	1						
	2	115	114	2		*Src1.Width*		***Src1.Width***		
	2	113	112	2	*Src1.ChanSel[7:4]*	*Src1.HorzStride*	***Src1.ChanSel[7:4]***	***Src1.HorzStride***		
	1	111	111	1	*Src1.AddrMode*		***Src1.AddrMode***			
	2	110	109	2	*Src1.SrcMod*		***Src1.SrcMod***			
	3	108	106	3	*Src1.RegNum [7:0]*		***Src1.AddrSubRegNum***			
	5	105	101	5						
	1	100	100	1	*Src1.SubRegNum [4]*	*Src1.SubRegNum [4:0]*	***Src1.AddrImm [9:4]***			
3	4	99	96	4	*Src1.ChanSel[3:0]*		***Src1.ChanSel[3:0]***	***Src1.AddrImm [9:0]***	*Imm[30:0]*	*Reg32*
	<u>6</u>	95	90	**0**						
	1	89	89	1	*FlagSubRegNum*					
	4	88	85	4	*Src0.VertStride*					
	1	84	84	1						
	2	83	82	2		*Src0.Width*		*Src0.Width*		
	2	81	80	2	*Src0.ChanSel[7:4]*	*Src0.HorzStride*	*Src0.ChanSel[7:4]*	*Src0.HorzStride*		
	1	79	79	1	*Src0.AddrMode*					
	2	78	77	2	*Src0.SrcMod*					
	3	76	74	3			*Src0.AddrSubRegNum*			
	5	73	69	5	*Src0.RegNum [7:0]*					
	1	68	68	1	*Src0.SubRegNum [4]*		*Src0.AddrImm [9:4]*			
2	4	67	64	4	*Src0.ChanSel[3:0]*	*Src0.SubRegNum [4:0]*	*Src0.ChanSel[3:0]*	*Src0.AddrImm [9:0]*	*Same*	
	1	63	63	1	*Dst.AddrMode*					
	2	62	61	2		*Dst.HorzStride*		*Dst.HorzStride*		
	3	60	58	3			*Dst.AddrSubRegNum*			
	5	57	53	5	*Dst.RegNum [7:0]*					
	1	52	52	1	*Dst.SubRegNum [4]*		*Dst.AddrImm [9:4]*			
	4	51	48	4	*Dst.ChanEn[3:0]*	*Dst.SubRegNum [4:0]*	*Dst.ChanEn[3:0]*	*Dst.AddrImm [9:0]*	*Same*	
	<u>1</u>	47	47	**0**						
	3	46	44	3	*Src1.SrcType*		***Src1.SrcType***			
	2	43	42	2	*Src1.RegFile*		***Src1.RegFile***			
	3	41	39	3	*Src0.SrcType*					
	2	38	37	2	*Src0.RegFile*					
	3	36	34	3	*Dst.DstType*					
1	2	33	32	2	*Dst.RegFile*				*Same*	
	1	31	31	1	*Saturate*					
	1	30	30	1	*DebugCtrl*				*Same*	
	<u>2</u>	29	28	**0**						
	4	27	24	4	*CondModifier*				*MDst.RegNum[8:5]*	
	3	23	21	3	*ExecSize*					
	1	20	20	1	*PredInv*					
	4	19	16	4	*PredCtrl*					
	2	15	14	2	*ThreadCtrl*					
	2	13	12	2	*ComprCtrl*					
	2	11	10	2	*DepCtrl*					
	1	9	9	1	*MaskCtrl*					
	1	8	8	1	*AccessMode*				*Same*	
	1	7	7	0	*(reserved for Opcode)*					
0	7	6	0	7	*Opcode*				*Same*	
	128			**111**	**TOTAL**					

13.2.1 Common Instruction Fields

As shown in Table 13-2, the meanings (encoding) of certain bit fields in the 128-bit instruction word varies depending on the values of other bit fields.

Table 13-3 provides the definition of common fields in the instruction word. The 'Width' column specifies the width of the field in bits. These common fields will be referred to later in describing the fields of different doublewords of the instruction. The definition for fields that have unique representation can be found in its corresponding doubleword of the instruction.

Table 13-3. Definitions of Common Instruction Fields

Field	Description	Width
CondModifier	**Conditional Modifier.** This field sets the flag register based on the internal conditional signals output from the execution pipe such as sign, zero, overflow, round-increment and NaNs, etc. If this field is set to 0000, no flag registers are updated. This field may also be referred to as the *flag destination control* field. This field applies to all instructions except *send*. 0000 = Do not modify the flag register (normal) 0001 = Zero or Equal ('**.z**' or '**.e**') 0010 = Not Zero or Not Equal ('**.nz**' or '**.ne**') 0011 = Greater-than ('**.g**') 0100 = Greater-than-or-equal ('**.ge**') 0101 = Less-than ('**.l**') 0110 = Less-than-or-equal ('**.le**') 0111 = Round-increment ('**.r**') 1000 = Overflow ('**.o**') 1001 = Unordered with Computed NaN ('**.u**') 1010 -1111 = Reserved	4
AddrMode	**Addressing Mode.** This field determines the addressing method of the operand. 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 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	1

Field	Description	Width
RegNum	**Register Number.** This field provides the register number for the operand. For GRF or MRF 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, if *RegFile* = **MRF**. 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 GEN4 Execution Environment chapter for details.	8
SubRegNum	**Sub-Register Number.** This field provides the sub-register number for the operand. For GRF or MRF 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 special encoding 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 *RegNum* field provides bits [12:5]. 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 = U5, if *RegFile* = **GRF** or **MRF** 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 GEN4 Execution Environment chapter for details.	5

Field	Description	Width
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. Format = U3 0x00 to 0x07 = Address Sub-Register number in the range of [0, 7]	3
AddrImm	**Address Immediate.** This field provides the immediate value in unit of byte to be added to the address register in order to compute the register address (byte-aligned region origin) for the operand. It is a 10-bit signed integer in 2's complement form. 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 numerical modification to 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 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)	2

Field	Description	Width
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 in power of 2, ranging from 0 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 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.* 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) 0110 = 32 Elements (applies to byte operand only) 0111-1110 = Reserved 1111 = **VxH or Vx1 mode** (only valid for register-indirect addressing in **Align1** mode)	4
Width	**Width.** 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	3

<table>
<tr><th>Field</th><th>Description</th><th>Width</th></tr>
<tr><td><b><i>HorzStride</i></b></td><td><b>Horizontal Stride.</b> 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><b><i>Imm32</i></b></td><td><b>32-bit Immediate.</b> 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.

Valid ranges according to data type:

<table>
<tr><th>Immediate Data Type</th><th>Valid Range (inclusive)</th></tr>
<tr><td>F</td><td>[0…±1.0*$2^{-128…127}$]</td></tr>
<tr><td>UW</td><td>[0, 65535]</td></tr>
<tr><td>W</td><td>[-32768, 32767]</td></tr>
<tr><td>UD</td><td>[0, 2^{32}-1]</td></tr>
<tr><td>D</td><td>[-2^{31}, 2^{31}-1]</td></tr>
<tr><td>VF</td><td>[0, ±0.125…±31]</td></tr>
<tr><td>V</td><td>[-8, 7]</td></tr>
</table></td><td>32</td></tr>
</table>

Field	Description	Width
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>.zywz:f*** would assign the channels as follows: Chan_0 = Data_2, Chan_1 = Data_1, Chan_2 = Data_3, Chan_3 = Data_2; Chan_4 = Data_6, Chan_5 = Data_5, Chan_6 = Data_7, Chan_7 = Data_6. 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
MsgDscpt31	**Message Description.** This field, containing 31-bit immediate values, provides the description of the message to be sent. This field only applies to the *send* 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	31
EOT	**End of Thread.** This field controls the termination of the thread. For a *send* 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 *send* instruction. It is not present for other instructions. 0 = The thread is not terminated 1 = **EOT**	1

13.2.2 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 *CurrDest.RegNum* field for the *send* instruction.

Table 13-4. Definitions of Fields in Operation Doubleword (DW0)

Bits	Description
31	**Saturate.** This field controls the destination saturation. 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 representable 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 numerical 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

Destination Type	Saturation Target Range (inclusive)
Float (F)	[0.0, 1.0]
Byte (UB)	[0, 255]
Signed Byte (B)	[-128, 127]
Word (UW)	[0, 65535]
Signed Word (W)	[-32768, 32767]
Double Word (UD)	[0, 2^{32}-1]
Signed Double (D)	[-2^{31}, 2^{31}-1]

Bits	Description
30	**DebugCtrl – Debug Control.** This field allows the insertion of a breakpoint at the current instruction. When the bit is set, hardware automatically stores the current IP in CR register and jumps to the System IP (SIP) BEFORE executing the current instruction. 0 = No breakpoint (normal) 1 = **"Breakpoint".** Breakpoint is inserted before this instruction is executed.
29	Reserved: MBZ
28	Reserved: MBZ

Bits	Description
27:24	**CondModifier** or **CurrDst.RegNum[3:0]** Definition of this bit field depends on whether the instruction is a *send* or not. Opcode != '*send*': **CondModifier:** This field sets the flag register based on the internal conditional signals output from the execution pipe. Opcode = '*send*': **CurrDst.RegNum[3:0]** This field sets the MRF register number for the current destination operand in the *send* instruction. No flag registers are updated for the *send* instruction. The 4-bit field provides full access of the 16 MRF registers. (See Instruction Reference chapter for *CurrDst*.)
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. 000 = 1 Channels (scalar operation) 001 = 2 Channels 010 = 4 Channels 011 = 8 Channels 100 = 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 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.

Nested table in the 27:24 row:

Opcode != '*send*'	Opcode = '*send*'
CondModifier: This field sets the flag register based on the internal conditional signals output from the execution pipe.	**CurrDst.RegNum[3:0]** This field sets the MRF register number for the current destination operand in the *send* instruction. No flag registers are updated for the *send* instruction. The 4-bit field provides full access of the 16 MRF registers. (See Instruction Reference chapter for *CurrDst*.)

Bits	Description
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. 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 **'.any4h'** 0111 = Predication with **'.all4h'** 1000 -1111 = Reserved Predicate Control in **Align1** access mode 0000 = No predication (normal) 0001 = Predication with sequential flag channel mapping 0010 = Predication with **.anyv** (any from f0.0-f0.1 on the same channel) 0011 = Predication with **.allv** (all of f0.0-f0.1 on the same channel) 0100 = Predication with **.any2h** (any in group of 2 channels) 0101 = Predication with **.all2h** (all in group of 2 channels) 0110 = Predication with **.any4h** (any in group of 4 channels) 0111 = Predication with **.all4h** (all in group of 4 channels) 1000 = Predication with **.any8h** (any in group of 8 channels) 1001 = Predication with **.all8h** (all in group of 8 channels) 1010 = Predication with **.any16h** (any in group of 16 channels) 1011 = Predication with **.all16h** (all in group of 16 channels) 1100 -1111 = Reserved

Bits	Description
15:14	**ThreadCtrl – Thread Control.** This field provides explicit control for thread switching. If this field is set to 00, it is up to the GEN4 execution units to manage thread switching. This is the normal operations mode. In this mode, for example, if the current instruction cannot proceed due to operand dependencies, EU switches to 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 also has instruction ready, 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) for the current thread is introduced for the thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. **Switch** is designed primarily as a safety feature in case there are race conditions for certain instructions. 00 = Normal Thread Control 10 = **"Switch"** 11 = Reserved
13:12	**ComprCtrl – Compression Control.** This field provides explicit control for instruction compression. If it is set to 00, the current instruction is a normal instruction (uncompressed) that can be directly dispatched for execution. A normal instruction most likely takes two instruction clock cycles to complete. If half of an architecture register (such as accumulator, mask registers and mask stack registers) is implicitly used by the instruction, the first half is used. If it is set to **SecHalf**, the current instruction is a normal instruction (uncompressed) that, if applicable, uses the second half of implicit architecture registers. If it is set to **Compr**, the current instruction is a compressed instruction that needs to be decompressed into two normal instructions. Hardware set the **ComprCtrl** field of the first decompressed instruction to 00 and that for the second decompressed instruction to **SecHalf**. The two native instructions are then dispatched for execution. A compressed instruction normally takes four instruction clock cycles to complete. 00 = Normal (uncompressed) instruction 01 = **"SecHalf"** 10 = **"Compr"** 11 = Reserved

Bits	Description
11:10	**DepCtrl – Destination Dependency Control.** This field selectively disables 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.** This field, together with **MaskCtrlEx** field, determines if conditional masks (A/B/L/CMasks) are used for creating the execution mask. Prediction control is still allowed. Encoding for the two-bit field [**MaskCtrlEx, MaskCtrl]** is as the following 00 = Use Masks (normal) 01 = **“NoMask”** – all masks are ignored for EMask creation for this instruction 10 = Reserved 11 = Reserved
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 13.3 for details of opcode encoding.

13.2.3 Instruction Destination Doubleword (DW1)

Destination Doubleword (DW1) contains the register file and numeric type of all operands, as well as the register region parameters of the destination operand. Table 13-5 shows the field definition of the Instruction Destination Doubleword. Furthermore, the Destination Register Region is described in Table 13-6 through Table 13-9.

Table 13-5. Instruction Destination Doubleword

Bits	Description
31:16	**Destination Register Region.** This word contains the parameters describing the register region of the destination operand. Subfield definition depends on the AccessMode. Detailed descriptions can be found in Table 13-6 through Table 13-9.
15	Reserved: MBZ
14:12	**Src1.SrcType – Source-1 Data Type.** This field specifies the numerical data type of the source operand <src1>. The bits of a source operand are interpreted as the identified numerical 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 numerical data types. Secondly, it has two 32-bit vector types – halfbyte integer vector (V) type and exponent-only float vector (VF) type. *Implementation Note 1: 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.* *Implementation Note 2: 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.* Source Register Type Encoding 000 = **“UD”**. **U**nsigned **D**oubleword integer 001 = **“D”**. Signed **D**oubleword integer 010 = **“UW”**. **U**nsigned **W**ord integer 011 = **“W”**. Signed **W**ord integer 100 = **“UB”**. **U**nsigned **B**yte integer 101 = **“B”**. Signed **B**yte integer 110 = Reserved 111 = **“F”**. Single precision **F**loat (32-bit) Source Immediate Type Encoding: 000 = **“UD”** 001 = **“D”** 010 = **“UW”** 011 = **“W”** 100 = Reserved 101 = **“VF”**. 32-bit restricted **V**ector **F**loat 110 = **“V”**. 32-bit halfbyte integer **V**ector 111 = **“F”**

Bits	Description
11:10	**Src1.RegFile – Source-1 Register File.** This field identifies the register file of source operand <src1>. 00 = **“ARF”**. Architecture Register File (**a**#, **acc**#, **f**#, **n**#, **null**, **ip**, etc.) 01 = **“GRF”**. General Register File (**r**#) 10 = **“MRF”**. Message Register File (**m**#) 11 = **“IMM”**. Immediate
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 numerical data type of the destination operand <dst>. The bits of the destination operand are interpreted as the identified numerical 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”**. **U**nsigned **D**oubleword integer 001 = **“D”**. Signed **D**oubleword integer 010 = **“UW”**. **U**nsigned **W**ord integer 011 = **“W”**. Signed **W**ord integer 100 = **“UB”**. **U**nsigned **B**yte integer 101 = **“B”**. Signed **B**yte integer 110 = Reserved 111 = **“F”**. Single precision **F**loat (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 = **“MRF”**. Message Register File (**m**#) 11 = **reserved**

The following tables describe the Destination Register Region based on the access mode and addressing mode.

Table 13-6. Destination Register Region in Direct + Align16 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. (See section 13.2.1 for definition of *AddrMode.*) For a *send* instruction, this field applies to **PostDst** – the post destination operand. Addressing mode for *CurrDst* (current destination operand) is fixed as Direct. (See Instruction Reference chapter for *CurrDst* and *PostDst*.)
14:13	Reserved: MBZ
12:5	**Dst.RegNum – Destination Register Number.** This field is the *RegNum* field for the destination operand. (See section 13.2.1 for definitions of *RegNum*.) For a *send* instruction, this field applies to **PostDst**.
4	**Dst.SubRegNum[4].** This is the 16-byte aligned sub-register address. (See section 13.2.1 for definitions of *SubRegNum*) For a *send* instruction, this field applies to **CurrDst**.
3:0	**Dst.ChanEn – Destination Channel Enable**. The channel enable field for the destination operand. (See section 13.2.1 for definitions of *ChanEn*) For a *send* instruction, this field applies to the **CurrDst**.

Table 13-7. Destination Register Region in Direct+Align1 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, it applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	**Dst.HorzStride – Destination Horizontal Stride.** This field is the *HorzStride* for the destination operand. For a *send* instruction, this field applies to **CurrDst**. **PostDst** only uses the register number.
12:5	**Dst.RegNum – Destination Register Number.** This field is the *RegNum* field for the destination operand. For a *send* instruction, this field applies to **PostDst**.
4:0	**Dst.SubRegNum – Destination Sub-Register Number.** This field is the **SubRegNum** for the destination operand. (See section 13.2.1 for definition of *SubRegNum*) For a *send* instruction, this field applies to **CurrDst**.

Table 13-8. Destination Register Region in Indirect+Align16 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, this field applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	Reserved: MBZ
12:10	**Dst.AddrSubRegNum – Destination Address Sub-Register Number.** This field is the *AddrSubRegNum* for the destination operand. (See section 13.2.1 for definition of *AddrSubRegNum.*) For a *send* instruction, this field applies to **PostDst**.
9:4	**Dst.AddrImm[9:4]** This is the half-register aligned *AddrImm* field for the destination operand. (See section 13.2.1 for definition of *AddrImm*) For a *send* instruction, this field applies to **PostDst**.
3:0	**Dst.ChanEn – Destination Channel Enable.** The channel enable field for the destination operand. For a *send* instruction, this field applies to the **CurrDst**.

Table 13-9. Destination Register Region in Indirect+Align1 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, this field applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	**Dst.HorzStride – Destination Horizontal Stride** This field is the *HorzStride* for the destination operand. For a *send* instruction, this field applies to **CurrDst**. **PostDst** only uses the register number.
12:10	**Dst.AddrSubRegNum – Destination Address Sub-Register Number.** This field is the *AddrSubRegNum* for the destination operand. For a *send* instruction, this field applies to **PostDst**.
9:0	**Dst.AddrImm – Destination Address Immediate.** This field is the byte-aligned *AddrImm* for the destination operand. For a *send* instruction, this field applies to **PostDst**.

13.2.4 Instruction Source-0 Doubleword (DW2)

Instruction Source-0 Doubleword (DW2) contains the first source operand and also flag register number.

- Table 13-10 shows the field definition for Direct Addressing with Align16.
- Table 13-11 shows the field definition for Direct Addressing with Align1.
- Table 13-12 shows the field definition for Indirect Addressing with Align16.
- Table 13-13 shows the field definition for Indirect Addressing with Align1.

Table 13-10. Instruction Source-0 Doubleword in Direct+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** 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.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This field is the *VertStride* for <src0> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	**Src0.ChanSel[7:4]** This is bits [7:4] of the *ChanSel* field for <src0> operand. (See section 13.2.1 for definition of *ChanSel*).It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This field is the *AddrMode* for <src0> operand. (See section 13.2.1 for definition of *AddrMode*) 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>. (See section 13.2.1 for definition of *SrcMod*)It is ignored if <src0> is an immediate operand.
12:5	**Src0.RegNum – Source-0 Register Number** This is the *RegNum* field for source operand <src0>. (See section 13.2.1 for definition of *RegNum*.) 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>. (See section 13.2.1 for definition of *SubRegNum*) It is ignored if <src0> is an immediate operand.
3:0	**Src0.ChanEn – Source-0 Channel Enable** This is the *ChanEn* field for source operand <src0>. (See section 13.2.1 for definitions of *ChanEn*) It is ignored if <src0> is an immediate operand.

Table 13-11. Instruction Source-0 Doubleword in Direct+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride** This is the *VertStride* field for <src0> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20:18	**Src0.Width.** This is the *Width* field for source operand <src0>. (See section 13.2.1 for definition of *Width*) It is ignored if <src0> is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand <src0>. (See section 13.2.1 for definition of *HorzStride*) It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 13.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 13.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:5	**Src0.RegNum – Source-0 Register Number.** This is the *RegNum* field for source operand <src0>. (See section 13.2.1 for definition of *RegNum*.) It is ignored if <src0> is an immediate operand.
4:0	**Src0.SubRegNum – Source-0 Sub-Register Number.** This is the *SubRegNum* field for source operand <src0>. (See section 13.2.1 for definition of *SubRegNum*) It is ignored if <src0> is an immediate operand.

Table 13-12. Instruction Source-0 Doubleword in Indirect+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This is the *VertStride* field for <src0> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	**Src0.ChanSel[7:4] – Source-0 Channel Select.** This is bits [7:4] of the *ChanSel* field for <src0> operand. (See section 13.2.1 for definition of *ChanSel*). It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 13.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 13.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:10	**Src0.AddrSubRegNum – Source-0 Address Sub-Register Number.** This is the *AddrSubRegNum* field for source operand <src0>. (See section 13.2.1 for definition of *AddrSubRegNum*.) It is ignored if <src0> is an immediate operand.
9:4	**Src0.AddrImm[9:4] – Source-0 Address Immediate.** This contains the half-register aligned *AddrImm* field ((bits [9:4]) for <src0>. (See section 13.2.1 for definition of *AddrImm*) It is ignored if <src0> is an immediate operand.
3:0	**Src0.ChanEn – Source-0 Channel Enable** . This is the *ChanEn* field for source operand <src0>. (See section 13.2.1 for definitions of *ChanEn*) It is ignored if <src0> is an immediate operand.

Table 13-13. Instruction Source-0 Doubleword in Indirect+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This is the *VertStride* field for <src0> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20:18	**Src0.Width.** This is the *Width* field for source operand <src0>. (See section 13.2.1 for definition of *Width*) It is ignored if <src0> is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand <src0>. (See section 13.2.1 for definition of *HorzStride*) It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 13.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 13.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:10	**Src0.AddrSubRegNum – Source-0 Address Sub-Register Number.** This is the *AddrSubRegNum* field for source operand <src0>. (See section 13.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src0> is an immediate operand.
9:0	**Src0.AddrImm – Source-0 Address Immediate.** This is the byte aligned *AddrImm* field for <src0>. (See section 13.2.1 for definition of *AddrImm*) It is ignored if <src0> is an immediate operand.

13.2.5 Instruction Source-1 Doubleword (DW3)

Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>). Table 13-14 and Table 13-15 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 MsgDescpt31.
  - — 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>, Table 13-16 and Table 13-17 define the fields in DW3 for indirectly addressed <src1> in Align16 and Align1 modes.

Table 13-14. Instruction Source-1 Doubleword in Direct + Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride.** This field is the *VertStride* for <src1> operand. (See section 13.2.1 for definition of *VertStride*) 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. (See section 13.2.1 for definition of *ChanSel*) 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. (See section 13.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.
12:5	**Src1.RegNum.** This field is the *RegNum* field for <src1> operand. (See section 13.2.1 for definition of *RegNum.*) It is ignored if <src1> is an immediate operand.
4	**Src1.SubRegNum[4].** This field is bit [4] of the *SubRegNum* field for <src1>. (See section 13.2.1 for definition of *SubRegNum*) It is ignored if <src1> is an immediate operand.
3:0	**Src1.ChanEn – Source-1 Channel Enable.** It is the channel enable field for <src1>. (See section 13.2.1 for definitions of *ChanEn*)It is ignored if <src1> is an immediate operand.

Table 13-15. Instruction Source-1 Doubleword in Direct + Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride.** This field is the *VertStride* for <src1> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20:18	**Src1.Width.** This is the *Width* field for source operand <src1>. (See section 13.2.1 for definition of *Width*) It is ignored if <src1> is an immediate operand.
17:16	**Src1.HorzStride.** This is the *HorzStride* field for source operand <src1>. (See section 13.2.1 for definition of *HorzStride*) 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. (See section 13.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.
12:5	**Src1.RegNum – Source-1 Register Number.** This is the *RegNum* field for source operand <src1>. (See section 13.2.1 for definition of *RegNum.*) It is ignored if <src1> is an immediate operand.
4:0	**Src1.SubRegNum – Source-1 Sub-Register Number.** This is the *SubRegNum* field for source operand <src1>. (See section 13.2.1 for definition of *SubRegNum*) It is ignored if <src1> is an immediate operand.

Table 13-16. Instruction Source-1 Doubleword in Indirect+Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride** This is the *VertStride* field for <src1> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20	Reserved: MBZ
19:16	**Src1.ChanSel[7:4] – Source-1 Channel Select** This is bits [7:4] of the *ChanSel* field for <src1> operand. (See section 13.2.1 for definition of *ChanSel*). It is ignored if <src1> is an immediate operand.
15	**Src1.AddrMode – Source-1 Address Mode** This is the *AddrMode* for source operand <src1>. (See section 13.2.1 for definition of *AddrMode*) 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>. (See section 13.2.1 for definition of *SrcMod*) 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>. (See section 13.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src1> is an immediate operand.
9:4	**Src1.AddrImm[9:4] – Source-1 Address Immediate** This contains the half-register aligned *AddrImm* field ((bits [9:4]) for <src1>. (See section 13.2.1 for definition of *AddrImm*) It is ignored if <src1> is an immediate operand.
3:0	**Src1.ChanEn – Source-1 Channel Enable** This is the *ChanEn* field for source operand <src1>. (See section 13.2.1 for definitions of *ChanEn*) It is ignored if <src1> is an immediate operand.

Table 13-17. Instruction Source-1 Doubleword in Indirect+Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride** This is the *VertStride* field for <src1> operand. (See section 13.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20:18	**Src1.Width** This is the *Width* field for source operand <src1>. (See section 13.2.1 for definition of *Width*) It is ignored if <src1> is an immediate operand.
17:16	**Src1.HorzStride** This is the *HorzStride* field for source operand <src1>. (See section 13.2.1 for definition of *HorzStride*) It is ignored if <src1> is an immediate operand.
15	**Src1.AddrMode – Source-1 Address Mode** This is the *AddrMode* for source operand <src1>. (See section 13.2.1 for definition of *AddrMode*) 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>. (See section 13.2.1 for definition of *SrcMod*) 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>. (See section 13.2.1 for definition of *AddrSubRegNum.*) 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>. (See section 13.2.1 for definition of *AddrImm*) It is ignored if <src1> is an immediate operand.

Table 13-18. GEN4 Compacted Instruction Format

DW#	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	Description
1	8	63	56	8	Bits[108:101] Source1 register number
	8	55	48	8	Bits [76:69] Source0 register number
	8	47	40	8	Bits [60:53] Destination register number
	5	39	35	5	Src1 Index[4:0]
	3	34	32	3	Src0 Index[4:2]
0	2	31	30	2	Src0 Index[1:0]
	1	29	29	1	CompactCtrl – Compaction Control
	1	28	28	1	Bit [89] FlagSubRegNum
	4	27	24	4	Bits [27:24] CondModifier
	1	23	23	1	MaskCtrlEx - Mask Control Extension
	5	22	18	5	Subregister Index[4:0]
	3	17	13	5	Datatype Index[4:0]
	3	12	8	5	Control Index[4:0]
	5	7	7	1	Bit[30] - Debug Control
	7	6	0	7	Bits[6:0] - Opcode

Table 13-19. Definitions of Fields in the Compact Instruction

Bits	Description
63:56	**Bits [108:101] Source1 register number** forms bits [108:101], the source 1 register number field.
55:48	**Bits [76:69] Source0 register number** This field, after unpacking, forms bits [76:69], the source 0 register number field, of the 128-bit instruction word.
47:40	**Bits [60:53] Destination register number** This field, after unpacking, forms bits [60:53], the destination register number field, of the 128-bit instruction word.
39:35	**Src1Index** The 5-bit index for source 1. The 12-bit table-look-up result forms bits [120:109], the source 1 register region fields, of the 128-bit instruction word
34:30	**Src0Index** The 5-bit index for source 0. The 12-bit table-look-up result forms bits [88:77], the source 0 register region fields, of the 128-bit instruction word.

Bits	Description
29	**CompactCtrl – Compaction Control** This field indicates whether the instruction is in the 64-bit compaction form. When this bit is set (bit 29 of DW0), the instruction length is only 64-bit.. The bit location is fixed in both 128-bit and 64-bit instruction forms. 0 = 128-bit form (normal) 1 = 64-bit compaction form
28	**Bit [89] – FlagSubRegNum** This field, after unpacking, is bit [89] of the 128-bit instruction word.
27:24	**Bits [27:24] – CondModifier** This field, after unpacking, is bits [27:24] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.
23	**MaskCtrlEx – Mask Control Extension.** This field is an extension of the **MaskCtrl** field, see **MaskCtrl** field for detailed definition. This field, after unpacking, is bit[28] of the 128-bit instruction word.
22:18	**SubRegIndex** The 5-bit index for sub-register fields. The 15-bit table-look-up result forms bits [100:96], [68,64] and [52,48] of the 128-bit instruction word.
17:13	**DataTypeIndex** The 5-bit index for data type fields. The 18-bit table-look-up result forms bits [63:61] and [46, 32] of the 128-bit instruction word.
12:8	**ControlIndex** The 5-bit index for data type fields. The 17-bit table-look-up result forms bits[31], and [23, 8] of the 128-bit instruction word.
7	**Bits [30] – DebugCtrl** This field, after unpacking, is bit [30] of the 128-bit instruction word.
6:0	**Bits [6:0] – Opcode** This field, after unpacking, is bits [6:0] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.

13.3 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.

There are total of 48 opcodes defined. These opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxb are reserved.

13.3.1 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 = arithmic). Bit 1 is always 0 (which is reserved for future extension to support rotation shift as 0 = shift, 1 = rotate). Bit 0 indicates the shift direction (0 = right, 1 = left).
- The opcodes for compare instructions (***cmp*** and ***cmpn***) share the common 6 MSBs in the form of 001000xb. Bit 0 indicates whether it is a normal compare, *cmp*, or a special compare-NaN, *cmpn*.
- This group of instructions does not implicitly update the accumulators.
- Instruction compression applies to this group.

Table 13-20. Move and Logic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
1	0x01	***mov***	Component-wise move	1	1
2	0x02	***sel***	Component-wise selective move based on predication	2	1
3	0x03	*Reserved*			
4	0x04	***not***	Component-wise one's compliment (bitwise not)	1	1
5	0x05	***and***	Component-wise logical AND (bitwise and)	2	1
6	0x06	***or***	Component-wise logical OR (bitwise or)	2	1
7	0x07	***xor***	Component-wise logical XOR (bitwise xor)	2	1
8	0x08	***shr***	Component-wise logical shift right	2	1
9	0x09	***shl***	Component-wise logical shift left	2	1
10-11	0x0A-0x0B	*Reserved*			
12	0x0C	***asr***	Component-wise arithmetic shift right	2	1
13-15	0x0D-0x0F	*Reserved*			
16	0x10	***cmp***	Component-wise compare, store condition code in destination	2	1
17	0x11	***cmpn***	Component-wise compare-NaN, store condition code in destination	2	1
20-31	0x12-0x1F	*Reserved*			

13.3.2 Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 13-21. Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	***jmpi***	Jump indexed	1	0
33	0x21	*Reserved*			
34	0x22	***if***	If	0/2	0
35	0x23	***iff***	Fast if	1	0
36	0x24	***else***	Else	1	0
37	0x25	***endif***	End if	0	0
38	0x26	***do***	Do	0	0
39	0x27	***while***	While	1	0
40	0x28	***break***	Break	1	0
41	0x29	***cont***	Continue	1	0
42	0x2A	***halt***	Halt	1	0
43	0x2B	*Reserved*			
44	0x2C	***msave***	Push mask to stack and update mask	1	1
45	0x2D	***mrest***	Pop mask stack and restore mask	1	1
46	0x2E	***push***	Push mask to stack without updating mask	1	1
46	0x2E	*Reserved*			
47	0x2F	***pop***	Pop mask stack without updating mask	2	0

13.3.3 Miscellaneous Instructions

This instruction group has an opcode format of 011xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 13-22. Miscellaneous Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
48	0x30	***wait***	Wait for (external) notification	1	0
49	0x31	***send***	Send	1	1
51-55	0x33-0x37	*Reserved*			
57-63	0x39-0x3F	*Reserved*			

13.3.4 Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

- The opcode for round instructions (***rndu***, ***rndd***, ***rnde***, and ***rndz***) share the common 5 MSBs in the form of 10001xxb, with the lower 2 bits indicate the type of round.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
  - Some instructions such as ***frc***, ***lzd***, etc., perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 13-23. Parallel Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
64	0x40	***add***	Component-wise addition	2	1
65	0x41	***mul***	Component-wise multiply	2	1
66	0x42	***avg***	Component-wise average of the two source operands	2	1
67	0x43	***frc***	Component-wise floating point truncate-to-minus-infinity fraction	1	1
68	0x44	***rndu***	Component-wise floating point rounding up (ceiling)	1	1
69	0x45	***rndd***	Component-wise floating point rounding down (floor)	1	1
70	0x46	***rnde***	Component-wise floating point rounding toward nearest even	1	1
71	0x47	***rndz***	Component-wise floating point rounding toward zero	1	1
72	0x48	***mac***	Component-wise multiply accumulate	2	1
73	0x49	***mach***	multiply accumulate high	2	1
74	0x4A	***lzd***	leading zero detection	1	1
75-79	0x4B-0x4F	*Reserved*			

13.3.5 Vector Arithmetic Instructions

- This instruction group has an opcode format of 101xxxxb.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
  - Some instructions such as ***dp4-dp2***, etc., perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 13-24. Vector Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
80	0x50	***sad2***	2-wide sum of absolute difference	2	1
81	0x51	***sada2***	2-wide sad accumulate	2	1
82-83	0x52-0x53	*reserved*			
84	0x54	***dp4***	4-wide dot product for 4-vector	2	1
85	0x55	***dph***	4-wide homogenous dot product for 4-vector	2	1
86	0x56	***dp3***	3-wide dot product for 4-vector	2	1
87	0x57	***dp2***	2-wide dot product for 4-vector	2	1
88	0x58	*reserved*			
89	0x59	***line***	Component-wise line equation computation (a multiply-add)	2	1
90	0x5A	*reserved*			
91	0x5B	*reserved*			
92-95	0x5C-0x5F	*reserved*			

13.3.6 Special Instructions

There are two special instructions, namely, *nop* (opcode = 0x7E) and *illegal* (opcode = 0x**00**).

- *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. It is useful for debugging purpose to signal when a thread execution hits a branch that erroneously causes the IP to reach an instruction memory location outside the program.
- *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.

Table 13-25. Special Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
0	0x00	***illegal***	Illegal instruction	0	0
96-124	0x60-0x7C	*Reserved*			
125	0x7D	*reserved*			
126	0x7E	***nop***	No-op	0	0
127	0x7F	*Reserved*	(may be used as an extension code)		

13.4 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.

13.4.1 Instruction Groups

<Instruction>	::=	<UnaryInstruction>
	|	<BinaryAccInstruction>
	|	<BinaryInstruction>
	|	<TriInstruction>
	|	<JumpInstruction>
	|	<BranchLoopInstruction>
	|	<ElseInstruction>
	|	<BreakInstruction>
	|	<MaskControlInstruction>
	|	<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**”
	|	“**sad2**” | “**sada2**” | “**dp4**” | “**dph**” | “**dp3**” | “**dp2**”
<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**”
<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**”
<MaskControlInstruction>	::=	<MaskPushOp> <MaskStackReg> <MaskReg>
	|	<MaskPopOp> <MaskReg> <MaskStackReg>
	|	<MaskPopOpEx> <MaskStackReg> <Imm32>
<MaskPushOp>	::=	“**msave**” | “**push**”
<MaskPopOp>	::=	“**mrest**”
<MaskPopOpEx>	::=	“**pop**”
<SyncInstruction>	::=	<Predicate> <SyncOp> <NotifyReg>
<SyncOp>	::=	“**wait**”
<SpecialInstruction>	::=	“**do**” | “**endif**” |“**nop**” | “**illegal**”

13.4.2 Destination Register

<Dst>	::=	<DstOperand>
	|	<DstOperandEx>
<DstOperand>	::=	<DstReg> <DstRegion> <WriteMask> <DstType>
<DstOperandEx>	::=	<AccReg> <DstRegion> <DstType>
	|	<FlagReg> <DstRegion> <DstType>
	|	<AddrReg> <DstRegion> <DstType>
	|	<MaskReg> <DstRegion> <DstType>
	|	<MaskStackReg>
	|	<ControlReg>
	|	<IPReg>
	|	<NullReg>
<DstReg>	::=	<DirectGenReg> | <IndirectGenReg>
	|	<DirectMsgReg> | <IndirectMsgReg>
<PostDst>	::=	<PostDstReg> <DstRegion> <WriteMask> <DstType>
	|	<NullReg>
<PostDstReg>	::=	<DirectGenReg> | <IndirectGenReg>
<CurrDst>	::=	<DirectAlignedMsgReg>

13.4.3 Source Register

Source with Accumulator Access and with Immediate

```
<SrcAccImm>                ::=   <SrcAcc>
                               | <Imm32> <SrcImmType>

<SrcAcc>                   ::=   <DirectSrcAccOperand>
                               | <IndirectSrcOperand>

<DirectSrcAccOperand>      ::=   <DirectSrcOperand>
                               | <SrcArcOperandEx>
                               | <AccReg> <SrcType>

<SrcArcOperandEx>          ::=   <FlagReg> <Region> <SrcType>
                               | <AddrReg> <Region> <SrcType>
                               | <MaskReg> <Region> <SrcType>
                               | <MaskStackReg>
                               | <ControlReg>
                               | <StateReg>
                               | <NotifyReg>
                               | <IPReg>
                               | <NullReg>

<IndirectSrcOperand>       ::=   <SrcModifier> <IndirectGenReg> <IndirectRegion>
                              <Swizzle > <SrcType>
```

Source without Accumulator Access

```
<Src>                      ::=   <DirectSrcOperand>
                               | <IndirectSrcOperand>

< DirectSrcOperand >       ::=   <SrcModifier> <DirectGenReg> <Region> <Swizzle>
                              <SrcType>
                               | <SrcArcOperandEx>

<TriSrc>                   ::=   <SrcModifier> <DirectGenReg> <Region> <Swizzle>
                              <SrcType>
                               | <NullReg>

<MsgDesc>                  ::=   <ImmDesc>
                               | <Reg32>
<Reg32>                    ::=   <DirectGenReg> <Region> <SrcType>
```

Source without Accumulator Access or IP Access

```
<SrcImm>                   ::=   <DirectSrcOperand>
                               | <Imm32> <SrcImmType>
```

13.4.4 Address Registers

```
<AddrParam>           ::=   <AddrReg> <ImmAddrOffset>
<ImmAddrOffset>       ::=   “”
                         |  “,” <ImmAddrNum>
```

13.4.5 Register Files and Register Numbers

```
<DirectGenReg>        ::=   <GenRegFile> <GenRegNum> <GenSubRegNum>
<IndirectGenReg>      ::=   <GenRegFile> “[“ <AddrParam> “]”
<GenRegFile>          ::=   “r”
<GenRegNum>           :: =  “0”…“127”
<GenSubRegNum>        :: =  “”
                         |  “.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>             ::=   “f0” <FlagSubRegNum>
<FlagSubRegNum>       :: =  “”
                         |   “.0”...“.1”

<MaskReg>             ::=   “Mask0” <MaskSubRegNum>
                         |  “AMask” | “IMask” | “LMask” | “CMask”
<MaskSubRegNum>       :: =  “”
                         |  “.0” … “.3”
```

```
<MaskStackReg>                 ::=    "ms0" <MaskStackSubRegNum>
                                  |   "ims" | "lms"
<MaskStackSubRegNum>           :: =   ""
                                  |   ".0" |  ".16"

<MaskStackDepthReg>            ::=    "MSD0" <MaskStackDepthSubRegNum>
                                  |   "IMSD" | "LMSD"
<MaskStackDepthSubRegNum>      :: =   ""
                                  |   ".0" … ".1"

<NotifyReg>                    ::=    "n" <NotifyRegNum>
<NotifyRegNum>                 :: =   "0"…"1"

<StateReg>                     ::=    "sr0" <StateSubRegNum>
<StateSubRegNum>               :: =   ".0"… ".1"

<ControlReg>                   ::=    "cr0" <ControlSubRegNum>
<ControlSubRegNum>             :: =   ".0" …".2"

<IPReg>                        ::=    "ip"

<NullReg>                      ::=    "null"
```

13.4.6 Relative Location and Stack Control

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

13.4.7 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"
```

13.4.8 Types

<SrcType>	::=	“**:f**” | “**:ud**” | “**:d**” | “**:uw**” | “**:w**” | “**:ub**” | “**:b**”
<SrcImmType>	::=	<SrcType> | “**:v**” | “**:vf**”
<DstType>	::=	<SrcType>

13.4.9 Write Mask

<WriteMask>	::=	“”
	|	“.” “**x**” | “.” “**y**” | “.” “**z**” | “.” “**w**”
	|	“.” “**xy**” | “.” “**xz**” | “.” “**xw**” | “.” “**yz**” | “.” “**yw**” | “.” “**zw**”
	|	“.” “**xyz**” | “.” “**xyw**” | “.” “**xzw**” | “.” “**yzw**”
	|	“.” “**xyzw**”

13.4.10 Swizzle Control

<Swizzle>	::=	“”
	|	“.” <ChanSel>
	|	“.” <ChanSel> <ChanSel> <ChanSel> <ChanSel>
<ChanSel>	::=	“**x**” | “**y**” | “**z**” | “**w**”

13.4.11 Immediate Values

<ImmAddrNum>	::=	“**-512**”… “**511**”
<Imm32>	::=	“**0.0**”… “**±1.0*$2^{-128…127}$**” | “**0**”…”**2^{32}-1**” | “**-2^{31}**”…”**2^{31}-1**”
<Imm16>	::=	“**0**”…”**2^{16}-1**” | “**-2^{15}**”…”**2^{15}-1**”
<ImmDesc>	::=	“**0**”…”**2^{32}-1**”

13.4.12 Predication and Modifiers

Instruction Predication

```
<Predicate>              ::=   “”
                           |   “(” <PredState> <FlagReg> <PredCntrl> “)”

<PredState>              ::=   “”
                           |   “+”
                           |   “-“
<PredCntrl>              ::=   “”
                           |   “.x” | “.y” | “.z” | “.w”
                           |   “.any2h” | “.all2h”
                           |   “.any4h” | “.all4h”
                           |   “.any8h” | “.all8h”
                           |   “.any16h” | “.all16h”
                           |   “.anyv” | “.allv”
```

Source Modification

```
<SrcModifier>            ::=   “”
                           |   “-”
                           |   “(abs)”
                           |   “-” “(abs)”
```

Instruction Modification

```
<ConditionalModifier>    ::=   “”
                           |   <CondMod> “. ” <FlagReg>
<CondMod>                ::=   “.z” | “.e” | “.nz” | “.ne” | “.g” | “.ge” | “.l” | “.le” | “.o” | “.r” | “.u”
<Saturate>               ::=   “”
                           |   “.sat”
```

Execution Size

```
<ExecSize>               ::=   “(“ <NumChannels> “)”
<NumChannels>            ::=   “1” | “2” | “4” | “8” | “16” | “32”
```

13.4.13 Instruction Options

```
<InstOptions>          ::=  “”
                         |  “{” <InstOption> “}”
                         |  “{” <InstOption> <InstOptionEx> “}”

<InstOptionEx>         ::=  “”
                         |  “,” <InstOption> <InstOptionEx>

<InstOption>           ::=  <AccessMode>
                         |  <ComprCtrl>
                         |  <ThreadCtrl>
                         |  <DependencyCtrl>
                         |  <MaskCtrl>
                         |  <DebugCtrl>
                         |  <SendCtrl>

<AccessMode>           ::=  “Align1” | “Align16”
<ComprCtrl>            ::=  “SecHalf” | “Compr” | “Compr4”
<ThreadCtrl>           ::=  “Switch” | “Atomic”
<DependencyCtrl>       ::=  “NoDDChk” | “NoDDClr”
<MaskCtrl>             ::=  “NoMask”| “NoCMask”
<DebugCtrl>            ::=  “Breakpoint”
<SendCtrl>             ::=  “EOT”
```

Note for Assembler: Compression control “Compr” has a direct map to the binary instruction word. It may be omitted as long as the Assembler is able to determine whether an instruction is in compressed mode or not based on the execution size and the mode of operation. However, “Compr4” requires Assembler to modify both compression control field and the MSB of the MRF destination register, so it must be present in the instruction if used.

Compr4 and ***NoCMask*** are not supported.

13.5 Deprecated Features

13.5.1 Defeatured Instructions

The following instructions are removed from Gen4 implementation mainly due to implementation cost/schedule reasons. They are candidates for future generations.

- Sum of Absolute Difference 4 (sad4)
- Sum of Absolute Difference Accumulate 4 (sada4)
- Add Accumulate (aac)
- Min (min)
- Max (max)
- Next (next)
- Swizzle (swz)
- Dot Product Accumulate 2 (dpa2)
- Rotation Shift Left (rsl)
- Rotation Shift Right (rsr)

13.5.2 Others

The following features are also deprecated from GEN4 implementation.

- Restricted 16-bit Half Floating-Point Numbers

§§

14 Instruction Set Reference

This chapter describes the functions of GEN4 instructions. Each GEN4 instruction is given a different page and the pages are sorted in alphabetical order according to assembly language mnemonic.

14.1 Conventions

14.1.1 Pseudo Code Format

The instructions are explained in the following pseudo-code format that resembles the GEN4 assembly instruction format.

```
[(<pred>)] opcode (<exec_size>) <dst> <src0> [<src1>]
```

Square brackets "[ ]" are used to signify that the field is optional. Saturation modifier and instruction options are omitted for simplicity.

14.1.2 General Macros and Definitions

INST_BYTE_COUNT is defined as a constant of 16 bytes.

```
#define INST_BYTE_COUNT 16 // byte count of instruction
word
```

Function *floor()* converts a floating point value to an integral floating point value. For a given floating point value, from its closest two integral float values, function *floor()* returns the one that is closer to the 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)
}
```

Function *Condition()* takes the conditional signals {SN, ZR, OF, IN, NC} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data.

```
Bool Condition(result, cmod)
{
}
```

Function *ConditionNaN()* takes the conditional signals {SN, ZR, OF, IN, NC, NS} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data. The only difference between *Condition()* and *ConditionNaN()* is that *ConditionNaN()* uses the NS (NaN of the second source) signal.

```
Bool ConditionNaN(result, cmod)
{
}
```

Function *Jump()* jumps the instruction sequence from the current instruction location by *InstCount* number of instructions. If *InstCount* is a positive number, it jumps forward; if *InstCount* is a negative number, it jumps backward; if *InstCount* is zero, it is effectively an infinite loop on the current instruction.

```
void Jump(int InstCount)
{
      IP = IP + (InstCount * INST_BYTE_COUNT)
}
```

14.1.3 Mask Stack Operations

The following operations manipulate mask and mask stack. They are used in the instruction pseudocodes.

Function *Reset()* resets both the depth count and stack adder of mask stack *DstStack*.

```
void DstStack.Reset()
{
      DstStack.DepthCount = 0;
      for (n = 0; n <= 15; n++) {
      DstStack.adder[n] == 0;
      }
}
```

Function *Push()* pushes mask *SrcMask* into mask stack *DstStack*. It also increments the depth count. If the depth count overflows (wrapped to 0), it signals the overflow exception for the ask stack. Mask *SrcMask* is unchanged. When SPF is set, nothing is done in this function.

```
void DstStack.Push(Mask SrcMask)
{
   if (!SPF) {
            DstStack.DepthCount++;
            if (DstStack.DepthCount == 0) {
                  DstStack.DepthOverflow_exception = TRUE;
         } else {
                  DstStack.DepthOverflow_exception = FALSE;
            }
            for (n = 0; n <= 15; n++) {
            if (SrcMask.BitPos(n) == 0) {
                        DstStack.adder[n]++;
            }
            }
      }
}
```

Function *Pop()* pops mask stack *DstStack* by *m* levels. It subtracts *m* from each stack counter. Any negative results are saturated to zero. The depth count is decremented by *m*. If *m* exceeds the current depth count, a stack underflow exception is signaled. When underflow occurs, depth count contains the modular value – amount of underflow. When SPF is set, nothing is done in this function.

```
void DstStack.Pop(unsigned int m)
{
   if (!SPF) {
            DstStack.DepthUnderflow_exception = (m >
      DepthCount);
            DstStack.DepthCount -= m;
            for (n = 0; n <= 15; n++) {
            if (DstStack.adder[n] > m) {
                        DstStack.adder[n] -= m;
            } else {
                        DstStack.adder[n] = 0;
            }
            }
      }
}
```

Function *TopOfStack()* returns the top entry of mask stack *SrcStack*.

```
Mask* SrcStack.TopOfStack()
{
   Mask* DestMask[15:0];
      for (n = 0; n <= 15; n++) {
      *DestMask.BitPos[n] = (SrcStack.adder[n] == 0);
      }
      Return DestMask;
}
```

Function *Evaluate()* the mask *Mask* according to control parameters in *Ctrl*. *Mask* may be the intermediate predication mask *PMask*, the intermediate branch mask *BMask*, or the internal execution mask *EMask*.

```
void Evaluate(Mask* Mask, int Ctrl)
{
   if (Mask == PMask) {
      if (Predication is off) {
         PMask[15:0] = 0xFFFF;
      } else {
         PMask[15:0] is set on predication control
         Details are omitted
      }
   } else if (Mask == BMask) {
      BMask[15:0]  = AMask[15:0];
            BMask[15:0] &= LMask[15:0];
            BMask[15:0] &= CMask[15:0];
   } else if (Mask == EMask) {
      EMask[15:0]  = AMask[15:0];
            EMask[15:0] &= IMask[15:0];
            EMask[15:0] &= LMask[15:0];
            If (Ctrl != NoCMask) {  // used for while
         EMask[15:0] &= CMask[15:0];
      }
      If (MaskCtrl == NoMask) {
         EMask[15:0] = 0xFFFF;
      }
            If (Ctrl != NoPMask) {  // used for 'sel'
         Evaluate(PMask);
                  EMask[15:0] &= PMask[15:0];
            }
      If (CompCtrl == SecHalf) {
         EMask[7:0] = EMask[15:8];
            }
            EMask[15:0] &= ExecMask(ExecSize);
            EMask[15:0] &= DestMask(ExecSize);
   }
}
```

Function *ExecMask()* returns a mask value for the execution size *ExecSize*. The mask bits of the first *ExecSize* channels are set to 1 and the rest bits are set to zero. This function is used by function *Evaluate()*.

```
Mask ExecMask(int ExecSize)
{
   Mask TmpMask[15:0] = 0xFFFF;
   for (int i=0; i < ExecSize; i++) {
      TmpMask[i] = 1;
   }
   Return TmpMask[15:0];
}
```

Function *DeskMask()* returns a mask value based on the destination channel select – part of the instruction word in **Align16** mode. It returns a 0xFFFF mask value if the instruction is in **Align1** mode. This function is used by function *Evaluate()*.

```
Mask DestMask()
{
   Mask TmpMask[15:0] = 0x0000;
   If (Dest.ChannelSelect == TRUE) {
      If (Dest.ChannelSelect[x] == TRUE) {
               TmpMask[0] = TmpMask[4] = 1;
               TmpMask[8] = TmpMask[12] = 1;
         }
      If (Dest.ChannelSelect[y] == TRUE) {
               TmpMask[1] = TmpMask[5] = 1;
               TmpMask[9] = TmpMask[13] = 1;
         }
      If (Dest.ChannelSelect[z] == TRUE) {
               TmpMask[2] = TmpMask[6] = 1;
               TmpMask[10] = TmpMask[14] = 1;
         }
      If (Dest.ChannelSelect[w] == TRUE) {
               TmpMask[3] = TmpMask[7] = 1;
               TmpMask[11] = TmpMask[15] = 1;
         }
   } else {
      TmpMask[15:0] = 0xFFFF;
      }
   Return TmpMask[15:0];
}
```

14.2 Instruction Description

The rest of the chapter contains the description of GEN4 instructions.

14.2.1 add – Addition

Opcode	Instruction	Description
64 (0x40)	add <dst> <src0> <src1>	Component-wise addition of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] add[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] add[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] add[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] + src1.chan[n];
   }
}
```

Description:

The *add* instruction performs component-wise addition of <src0> and <src1> and stores the results in <dst>.

Addition of two floating point numbers follows rules in Table 14-1 (or Table 14-2), if the current floating point mode is IEEE mode (or ALT mode).

Table 14-1. Floating point addition of A (column) and B (row) in IEEE mode

	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
–inf	–inf	–inf	–inf	–inf	–inf	–inf	–inf	NaN	NaN
–finite	–inf	*	A	A	A	A	**	+inf	NaN
–denorm	–inf	B	–0	–0	+0	+0	B	+inf	NaN
–0	–inf	B	–0	–0	+0	+0	B	+inf	NaN
+0	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+denorm	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+finite	–inf	**	A	A	A	A	***	+inf	NaN
+inf	NaN	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

*	Result can be { –finite}
**	Result can be {–finite, –0, +0, +finite}
***	Result can be { +finite}

Table 14-2. Floating point addition of A (column) and B (row) in ALT mode

	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–finite	+0	▨
–finite	–fmax	*	A	A	A	A	**	+fmax	▨
–denorm	–fmax	B	–0	–0	+0	+0	B	+fmax	▨
–0	–fmax	B	–0	–0	+0	+0	B	+fmax	▨
+0	–fmax	B	+0	+0	+0	+0	B	+fmax	▨
+denorm	–fmax	B	+0	+0	+0	+0	B	+fmax	▨
+finite	–finite	**	A	A	A	A	***	+fmax	▨
+fmax	+0	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	▨
****	▨	▨	▨	▨	▨	▨	▨	▨	▨

Notes:

*	Result can be { –fmax, –finite}
**	Result can be {–finite, –0, +0, +finite}
***	Result can be { +fmax, +finite}
****	Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

14.2.2 and – Logical And

Opcode	Instruction	Description
5 (0x05)	and <dst> <src0> <src1>	Performing component-wise logic AND of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] and[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] and[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] and[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] & src1.chan[n];
   }
}
```

Description:

The *and* instruction performs component-wise logic AND operation between <src0> and <src1> and stores the results in <dst>. Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction only applies to integer data types. The behavior is undefined if any operand is float.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.

- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

14.2.3 asr – Arithmetic Shift Right

Opcode	Instruction	Description
12 (0x0C)	asr <dst> <src0> <src1>	Performing component-wise arithmetic right shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] asr[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.channel[n] == 1) {
      if (src0.chan[n] >= 0) {
         dst.chan[n] = src0.chan[n] >> src1.chan[n];
      } else {
         int maskLSB = pow(2, src1.chan[n]) – 1;
         if (maskLSB & src0.chan[n] == 0) {
            dst.chan[n] = sign(src0.chan[n]) *
                                  ((abs)src0.chan[n] >>
                           src1.chan[n]);
         } else {
            dst.chan[n] = sign(src0.chan[n]) *
                                  ((abs)src0.chan[n] >>
                           src1.chan[n])–1;
         }
      }
   }
}
```

Description:

The *asr* instruction performs component-wise arithmetic right shift of <src0> and storing the results in <dst>. Arithmetic right shift performs sign-extension by repeating the MSB of each data channel of <src0>. The amount of bit shift is provided by <src1>. Only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The rest of MSBs of <src1> data channels are ignored.

Operands for this instruction can be signed or unsigned integers, but cannot be floating point type. 5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction is effectively a power-of-2 integer divide with truncate in 2's complement form. Truncate in 2's complement form is also known as downward rounding – closest integer that is smaller than or equal to the result. For example, regardless of the bit shift amount in <src1>, the result of arithmetic right-shift of -1 (<src0>) is always -1.

Restrictions:

This instruction does not work with float type operands.

The results are not stored in the accumulator register and therefore this instruction cannot have accumulator as destination.

14.2.4 avg – Average

Opcode	Instruction	Description
66 (0x42)	avg <dst> <src0> <src1>	Component-wise averaging of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] avg[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = (src0.chan[n] + src1.chan[n] + 1) >> 1;
   }
}
```

Description:

The *avg* instruction performs component-wise integer average of <src0> and <src1> and stores the results in <dst>. An integer average uses integer upward rounding. It is equivalent to increment one to the addition of <src0> and <src1> and then apply an arithmetic right shift to this intermediate value.

Restrictions:

This instruction only applies to integer data types. The behavior is undefined if any operand is float.

14.2.5 break – Break

Opcode	Instruction	Description
40 (0x28)	Break <exitcode>	Terminating enabled execution channels and conditionally breaking out from the inner most loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] break (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] break (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
LMask = LMask & !EMask;
CMask = CMask & !Emask;          // *** (1)
TempMask = LMask & AMask;           // *** (2)
if (TempMask == 0) {
        CMask = Lstack.TopOfStack;
    LStack.pop(1);
    LMask = Lstack.TopOfStack;
    LStack.pop(1);
        if (<exitcode.ISPopCount> != 0) {

                IStack.pop(<exitcode.ISPopCount> - 1);
        IMask = IStack.TopOfStack;
        IStack.pop(1);
    }
    Jump(<exitcode.IPCount>);
}
```

Description:

The *break* instruction is used to early-out from the inner most loop. When executed, the *break* instruction terminates the loop for all enabled execution channels. This is performed by clearing the associated bit positions in LMask. The loop is complete when all bit positions of LMask are cleared. If the loop completes on a *break* instruction, the specified number of entries (If-nesting depth) is popped from the IStack, 2 levels of pop of the LStack, and a jump taken to the address specified in the <exitcode> field. These pops are intended to restore the mask-stacks and LMask/CMask/IMask to their values prior to entering the loop.

The following table describes the 32-bit exit code <exitcode>. The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction following loop block. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:20	Reserved: MBZ
19:16	**ISPopCount (If-Stack Pop Count)** This field specifies the number of times to pop the IStack. Format = U4. Unsigned integer [0, 15]
15:0	**InstCount (Jump Instruction Count)** This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When the condition is true, this instruction restores LMask and CMask. If "ISPopCount" is not zero, it also restores IMask.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Notes to the pseudo code:

(1) CMask is updated by a break instruction to force it to always be a 1's-subset of the LMask. This allows for the sharing of a mask-stack and the reliance on a fixed order of pushing LMask first and then CMask for embedded loops. If we did not update CMask, the order of pushing would be data dependent and would take significant code to do properly or we would not be able to share a stack.

(2) It should not take CMask as the subset of channels disabled by CMask are temporally terminated but still in the do-while loop.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

14.2.6 cmp – Compare

Opcode	Instruction	Description
16 (0x10)	cmp.<cmod> <dst> <src0> <src1>	Component-wise comparison of <src0> and <src1> according to conditional modifier in <cmod> and storing the results in flag register in <cmod> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] cmp[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] cmp[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] cmp[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      bitMask[n] = 0;
      if (WrEn.chan[n] == 1) {
            results[n] = src0.chan[n] - src1.chan[n];
            bitMask[n] = Condition(results[n]);
            dst.chan[n][0] = bitMask[n];
      }
}
flag# = bitMask;
```

Description:

The *cmp* instruction performs component-wise comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional code (excluding NS signal) based on the conditional modifier, and storing the conditional bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it is not null, for the enabled channels, the LSB of the result in the destination channel contains the flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by 'and-ing' an 0x01 constant).

If <exec_size> is equal or less than 8, when '*SecHalf*' option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmp* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

14.2.7 cmpn – Compare NaN

Opcode	Instruction	Description
17 (0x11)	cmpn.<cmod> <dst> <src0> <src1>	Performing component-wise special NaN comparison of <src0> and <src1> according to conditional modifier in <cmod> and storing the results in flag register in <cmod> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] cmpn[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] cmpn[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] cmpn[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      bitMask[n] = 0;
      if (EMask.chan[n] == 1) {
            results[n] = src0.chan[n] - src1.chan[n];
            bitMask[n] = ConditionNaN(results[n]);
            dst.chan[n][0] = bitMask[n];
      }
}
flag# = bitMask;
```

Description:

The *cmpn* instruction performs component-wise special-NaN comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional signals including NS based on the conditional modifier, and storing the conditional flag bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it is not null, for the enabled channels, the LSB of the result in the destination channel contains the flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by 'and-ing' an 0x01 constant).

If <exec_size> is equal or less than 8, when '*SecHalf*' option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmpn* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

This instruction is similar to *cmp*. The only difference is that if the second source operand <src1> is a NaN, the result for any conditional modifier except **.nz** is false.

For integer operands, *cmpn* and *cmp* are identical.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

14.2.8 cont – Continue

Opcode	Instruction	Description
41 (0x29)	cont <exitcode>	Temporally disabling enabled execution channels for the remainder of the inner most loop and conditionally jumping to the last instruction (while) of the loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] cont (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] cont (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
CMask = CMask & !EMask;
TempMask = CMask & AMask;           // *** (1)
if (TempMask == 0) {
   if (<exitcode.ISPopCount> != 0) {
            IStack.pop(<exitcode.ISPopCount> - 1);
      IMask = IStack.TopOfStack;
      IStack.pop(1);
   }
   Jump(<exitcode.IPCount>);
}
```

Description:

The *cont* instruction disables execution for the subset of channels for the remainder of the current loop iteration. Channels remain disabled until explicitly re-enabled by the thread through writing to the CMask register. If CMask = 0 after evaluating this instruction, a jump is made a distance of <InstCount> where execution continues. It is expected that this is the location of the associated 'while'. For proper loop operation, CMask must be re-initialized to EMask on each loop pass.

The following table describes the 32-bit exit code <exitcode>. The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should point to the loop's associated 'while' instruction. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:20	Reserved: MBZ
19:16	**ISPopCount (If-Stack Pop Count)** This field specifies the number of times to pop the IStack. Format = U4. Unsigned integer [0, 15]
15:0	**InstCount (Jump Instruction Count)** This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When the condition is true and "ISPopCount" is not zero, this instruction restores IMask.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Upon completion of this instruction, B/LMask will only be valid if popping 1 level. If more than one level needs to be popped from any mask-stack, n-1 levels should be popped at the instruction and the final level popped by an explicit *pop* instruction.

Note that the 'break' instruction also updates CMask. This allows CMask and LMask to share the same mask-stack.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

14.2.9 do – Do

Opcode	Instruction	Description
38 (0x26)	do	Updating mask and mask stack to enter a do-while loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
do
```

Syntax:

```
do
```

Pseudocode:

```
Evaluate(EMask);
LStack.push(LMask);
LStack.push(CMask);
LMask = EMask;
CMask = EMask;
```

Description:

The *do* instruction indicates the start of a do-while block. Each *do* must have a matching *while* instruction. Execution of the *do* instruction causes the LMask and CMask (in that order) to be saved to the LStack for preservation and eventual restoration upon completion of the do-while block.

This instruction is equivalent to two *msave* instructions (in the order of "*msave lstack lmask*" and "*msave lstack cmask*"). It is an efficient construct for a do-while block.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

SPF effectively turns this instruction into a nop, as LMask and CMask should be coherent with EMask. It may be used as instruction filler for code readability keeping in mind that a nop wastes an instruction cycle.

Restrictions:

Predication is not allowed. Instruction compression does not apply to this instruction.

Execution size is ignored for this instruction.

14.2.10 dp2 – Dot Product 2

Opcode	Instruction	Description
87 (0x57)	Dp2 <dst> <src0> <src1>	Performing two-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] dp2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
   fTmp = src0.chan[n] * src1.chan[n]
            + src0.chan[n+1] * src1.chan[n+1];
      if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
      if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
      if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
      if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp2* instruction performs a two-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every third and fourth elements of <src0> (post-source-swizzle if present) are not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of four channels is 0.0f, the sign of the result correctly reflects the input data of the first two channels of the group of four.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

14.2.11 dp3 – Dot Product 3

Opcode	Instruction	Description
86 (0x56)	dp3 <dst> <src0> <src1>	Performing three-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] dp3[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
   fTmp = src0.chan[n] * src1.chan[n]
            + src0.chan[n+1] * src1.chan[n+1]
            + src0.chan[n+2] * src1.chan[n+2];
      if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
      if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
      if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
      if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp3* instruction performs a three-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every fourth element of <src0> (post-source-swizzle if present) is not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of 4 channels is 0.0f, the sign of the result correctly reflects the input data of the first three channels of the group of four.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

14.2.12 dp4 – Dot Product 4

Opcode	Instruction	Description
84 (0x54)	dp4 <dst> <src0> <src1>	Performing four-wide dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] dp4[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
   fTmp = src0.chan[n] * src1.chan[n]
            + src0.chan[n+1] * src1.chan[n+1]
            + src0.chan[n+2] * src1.chan[n+2]
            + src0.chan[n+3] * src1.chan[n+3];
      if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
      if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
      if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
      if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp4* instruction performs a four-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

14.2.13 dph –Dot Product Homogeneous

Opcode	Instruction	Description
85 (0x55)	dph <dst> <src0> <src1>	Performing four-wide homogeneous dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] dph[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
   fTmp = src0.chan[n] * src1.chan[n]
            + src0.chan[n+1] * src1.chan[n+1]
            + src0.chan[n+2] * src1.chan[n+2]
            + src1.chan[n+3];
      if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
      if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
      if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
      if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dph* instruction performs a four-wide homogeneous dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every fourth element of <src0> (post-source-swizzle if present) is forced to 1.0f.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

14.2.14 else – Else

Opcode	Instruction	Description
36 (0x24)	else <exitcode>	An optional statement within an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
else (<exec_size>) <exitcode>
```

Syntax:

```
else (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
IMask = !IMask & IStack.TopOfStack;
Evaluate(EMask);            // *** (1)
if (EMask == 0) {
        IMask = IStack.TopOfStack;
    IStack.pop(ISPopCount);
    Jump(<exitcode.IPCount>);
}
```

Description:

The *else* instruction is an optional statement within an ***if/else/endif*** block of code. It restricts execution within the else/endif portion to the opposite set of channels enabled under the if/else portion. Channels which were inactive prior to entering the ***if***/***endif*** block remain inactive throughout the entire block.

The IMask, which maintains the bit-mask of enabled channels inside the conditional block, is updated to reflect the new subset of active channels within the ***else/endif*** portion. If all channels are inactive (IMask = 000…), a relative jump is performed to the location specified in <exitcode.IPCount>. The jump target should be the instruction immediately following the matching ***endif*** instruction for that conditional block.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). <exitcode> must be an immediate operand, whereas IPCount is a signed

16-bit number and is intended to be forward referencing. This value is added to IP pre-increment. The ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack. Hardware behavior is undefined if any other values are used in this field.

Bit	Description
31:20	Reserved: MBZ
19:16	**ISPopCount (If-Stack Pop Count)** This field specifies the number of times to pop the IStack. Format = U4. ***Must be set to 1***.
15:0	**InstCount (Jump Instruction Count)** This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Each ***else*** instruction must be preceded with a matching ***if*** instruction and followed by a matching ***endif*** instruction.

Pseudocode notes:

(1) Uses the updated IMask calculated during instruction execution, not the IMask prior to instruction dispatch.

(2) The EMask calculation already has <exec_size> incorportated; therefore the full 16-bit EMask value can be used in the comparison.

Restrictions:

Instruction compression is not allowed.

Predication is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

14.2.15 endif – End-If

Opcode	Instruction	Description
37 (0x24)	endif <exitcode>	Restoring execution to those data channels that were active prior to the if/else/endif block.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
endif <exitcode>
```

Syntax:

```
endif imm32
```

Pseudocode:

```
IMask = IStack.TopOfStack;
IStack.pop(ISPopCount);
```

Description:

The *endif* instruction terminates an if/else/endif block of code. It restores the execution to these data channels that were active prior to the if/else/endif block by popping the IMask once. IMask is updated to reflect the popped value from the IStack.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). <exitcode> must be an immediate operand, whereas ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack.

Software must put a 1 in unsigned integer format in this field as this instruction only pop the IStack once. Hardware behavior is undefined if any other values are used in this field.

<exitcode> must be an immediate operand, whereas ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack.

Bit	Description
31:20	Reserved: MBZ
19:16	**ISPopCount (If-Stack Pop Count).** This field specifies the number of times to pop the IStack. Format = U4. ***Must be set to 1***.
15:0	Reserved: MBZ

This instruction updates IMask, therefore, a thread switch is always invoked after the issuance of an 'endif' instruction.

If SPF is set, this instruction restores IMask from IStack's Top-Of-Stack, which is supposed to be static for code with single program flow.

Execution size is ignored for this instruction.

Restrictions:

Instruction compression does not apply to this instruction.

Predication is not allowed.

14.2.16 frc – Fraction

Opcode	Instruction	Description
67 (0x43)	frc <dst> <src0>	Taking component-wise truncate-to-minus-infinity fraction operation of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•		•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] frc[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] frc[.<cmod>] (<exec_size>) reg reg
[(<pred>)] frc[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] – floor(src0.chan[n]);
   }
}
```

Description:

The *frc* instruction computes, component-wise, the truncate-to-minus-infinity fractional values of <src0> and stores the results in <dst>. The results, in the range of [0.0, 1.0], are the fractional portion of the source data.

Source operand for this instruction must be of floating point type. This instruction can only operate on normalized floating source and therefore cannot take accumulator as source or destination operand.

This instruction only applies to floating point operands.

Floating point fraction computation follows rules in Table 14-3 (or Table 14-4), if the current floating point mode is IEEE mode (or ALT mode).

Table 14-3. Floating point fraction computation in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	NaN	*	+0	+0	+0	+0	*	NaN	NaN

Notes:

* Result is in the range of [+0, 1) – not including 1.

Table 14-4. Floating point fraction computation in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	**
<dst>	+0	*	+0	+0	+0	+0	*	+0	▨

Notes:

* Result is in the range of [+0, 1) – not including 1.

** Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

Saturation modifier does not apply to this instruction.

This instruction cannot take accumulator as source or destination operand as it can only operate on normalized floating source.

This instruction does implicitely update accumulator register when enabled, however with undefined values.

14.2.17 halt – Halt

Opcode	Instruction	Description
42 (0x2A)	halt <exitcode>	Temporarily suspending execution for all enabled execution channels.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] halt (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] halt (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
AMask = AMask & !EMask;
if (AMask == 0) {
   Jump(<exitcode.IPCount>);
}
```

Description:

The *halt* instruction temporarily suspends execution for all enabled compute channels. The value of AMask is updated, with bits in positions of enabled channels set to '0'. If all the bits of the resultant AMask are cleared, a jump is made <inst_count> instructions away.

The *halt* instruction is also used inside subroutines as a 'return', utilizing AMask to keep track of which execution channels have returned and which to continue execution. Since there is no hardware mask stack for AMask, software must manually preserve the value of AMask around a subroutine call.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should typically point to code which restores AMask.

Bit	Description
31:16	Reserved: MBZ
15:0	**InstCount (Jump Instruction Count)** This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

The jump target used in the *halt* instruction typically crosses loops or 'if's, leaving the associated mask-stacks stale. It is the responsibility of program to manually restore these stacks to their values at the nesting level of the jump target.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

As DMask is not automatically reloaded into AMask upon completion of this instruction, software has to manually restore AMask upon completion.

14.2.18 if – If

Opcode	Instruction	Description
34 (0x22)	if <exitcode>	Signifying the start of an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] if (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] if (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
IStack.push(IMask);
IMask = EMask;
if (EMask == 0) {
        Jump(<exitcode.IPCount>);
}
```

Description:

The *if* instruction starts an if/endif or an if/else/endif block of code. It restricts execution within the conditional block to only those channels that were enabled via EMask calculation in this instruction.

Each *if* instruction must have a matching *endif* instruction and may have up to one matching *else* instruction before *endif*.

The *if* instruction causes the value of the IMask prior to the *if* instruction to be saved to the IStack for retrieval at the conclusion of the conditional block. The *if* instruction evaluates the EMask. If all channels are inactive (for the if/endif or if/else block), a jump is performed of the relative distance as specified in the instruction. This jump must be to the matching *else* instruction when present, or otherwise to the matching *endif* instruction of that conditional block.

The following table describes the 32-bit exit code <exitcode>. <exitcode> must be an immediate operand, whereas IPCount is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN4 instruction binary,

<exitcode> is at location <src1> and must be of type D (signed doubleword integer). IP register must be put (for example, by the assembler) at both <dst> and <src0> locations.

Bit	Description
31:16	Reserved: MBZ
15:0	**IPCount (Jump Instruction Count).** This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Restrictions:

Instruction compression is not allowed.

14.2.19 iff – Fast-If

Opcode	Instruction	Description
35 (0x23)	iff <exitcode>	Signifying the start of a fast if/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] iff (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] iff (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask);
if (EMask != 0) {
      IStack.push(IMask);
      IMask = EMask;
} else {
      Jump(<exitcode.IPCount>);
}
```

Description:

The *iff* instruction starts a high-performance (fast-if) if/endif block of code. It restricts execution within the conditional block to only those channels that were enabled via EMask calculation by this instruction.

Each *iff* instruction must have a matching *endif* instruction, but must NOT have a matching *else* instruction.

This fast-if has only one execution cycle of overhead comparing 2-3 for the general purpose *if* instruction. Unlike the *if* instruction, this instruction only pushes the IMask when at least one execution channels is active (i.e. EMask != 0) and thus requires processing within the if-endif block. When all execution channels are inactive (i.e. EMask = 0), the mask stack push is not performed, and execution continues at a relative distance specified by the <exitcode.IPCount> field. In this case, since the mask stack push was not performed, the mask stack pop associated with an *endif* must not be executed. Thus it is intended that the instruction pointed to by <exitcode.IPCount> is one instruction beyond the block's *endif* instruction.

The following table describes the 32-bit exit code <exitcode>. <exitcode> must be an immediate operand, whereas IPCount is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN4 instruction binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). IP register must be put (for example, by the assembler) at both <dst> and <src0> locations.

Bit	Description
31:16	Reserved: MBZ
15:0	**IPCount (Jump Instruction Count).** This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction may perform a mask-stack push/pop operation. If it is performed, mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Restrictions:

Instruction compression is not allowed.

Each *iff* must have one matching *endif*, but cannot have a matching *else* instruction.

14.2.20 jmpi – Jump Indexed

Opcode	Instruction	Description
32 (0x20)	jmpi <index>	Redirecting program execution to <index> instructions forward of the current post-incremented instruction pointer.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] jmpi (1) <exitcode> {NoMask}
```

Syntax:

```
[(<pred>)] jmpi (1) reg32 {NoMask}
[(<pred>)] jmpi (1) imm32 {NoMask}
```

Pseudocode:

```
Evaluate(WrEn);
if (WrEn != 0) {
    Jump(<exitcode.index> + 1);
}
```

Description:

The *jmpi* instruction redirects program execution to <exitcode.index> instructions forward of the current **post-incremented** instruction pointer. <exitcode.index> is treated a signed integer value, with positive integers or zero generating forward jumps, and negative integers generating backward jumps. An <exitcode.index> value of 0 means execution continues at the instruction immediately following the *jmpi* instruction, while an index value of -1 would imply an infinite loop.

Bit	Description
31:16	Reserved: MBZ
15:0	**index (Jump Index)** This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

<exitcode> may be a scalar register or an immediate. The data type of <exitcode> must be D (signed doubleword integer). However, hardware only uses lower 16 bits of <exitcode>. The valid range of <exitcode.index> is [−32768, 32767]. Behavior for <exitcode.index> outside that range is undefined.

This instruction executes regardless of the calculated WrEn at the time of issue. – To reduce hardware complexity, instruction optional control {NoMask} must be set for this instruction. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

Execution size must be 1.

Predication is allowed to provide conditional jump with a scalar condition. As the execution size is 1, the first channel of PMASK (flags post prediction control and negate) is used to determine whether the jump is taken or not. If the condition is false, the jump is not taken and the IP immediately following will be executed next.

In GEN4 binary, <exitcode.index> is at location <src1>. IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

Restrictions:

Instruction compression is not allowed.

14.2.21 line – Line

Opcode	Instruction	Description
89 (0x59)	Line <dst> <src0> <src1>	Computing a component-wise line equation (v = p*u+q) of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] line[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] line[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] line[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
for (n = 0; n < exec_size; n++) {
   dwP = src0.RegNum.SubRegNum[bits4:2]   // a DW aligned
scalar
   dwQ = src0.RegNum.(SubRegNum[bit4]|0x8)   // 4-th
component
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = dwP * src1.chan[n] + dwQ
   }
}
```

Description:

The *line* instruction computes a component-wise line equation (*v* = *p***u*+*q* where *u*/*v* are vectors and *p*/*q* are scalars) of <src0> and <src1> and storing the results in <dst>. <src1> is the input vector *u*. <src0> provides input scalars *p* and *q*, where *p* is the scalar value based on the region description of <src0> and *q* is the scalar value implied from <src0> region. Specifically, *q* is the fourth component of the 4-tuple (128-bit aligned) that *p* belongs to.

Restrictions:

This is a specialized instruction that only support execution size of 8 or 16.

<src0> region must be a replicated scalar (with HorzStride = VertStride = 0).

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

In particular, <src0> must be float. <src1> may be float, byte or word integer. <src1> cannot be dword integer. <dst> may be float or integer of any size.

Source operands cannot be an accumulator register.

<src0> for *line* instruction has to have .0 or .4 as the subregister number.

14.2.22 lzd – Leading Zero Detection

Opcode	Instruction	Description
74 (0x4A)	lzd <dst> <src0>	Performing component-wise leading zero detection of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•		[INT]	[INT]

Format:

```
[(<pred>)] lzd[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] lzd[.<cmod>] (<exec_size>) reg reg
[(<pred>)] lzd[.<cmod>] (<exec_size>) reg reg
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      UD udScalar = src0.chan[n];
      UD cnt = 0;
      while ( (udScalar & (1<<31)) == 0 && cnt != 32) {
         cnt ++;
         udScalar = udScalar << 1;
            }
      dst.chan[n] = cnt;
   }
}
```

Description:

The *lzd* instruction counts component-wise the leading zeros from <src0> and storing the resulting counts in <dst>.

This instruction only work on unsigned dword source. Source operand may be a signed or unsigned. If it is a signed integer, source modifier (abs) must be used to convert the source into an unsigned integer type.

The destination operand must also be of unsigned dword type.

Restrictions:

The destination operand cannot be the accumulator.

This instruction does implicitely update accumulator register when enabled, however with undefined values.

14.2.23 mac – Multiply Accumulate

Opcode	Instruction	Description
72 (0x48)	mac <dst> <src0> <src1>	Performing component-wise multiply accumulate of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] mac[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mac[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mac[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] * src1.chan[n] +
acc0.chan[n]
   }
}
```

Description:

The *mac* instruction takes component-wise multiplication of <src0> and <src1>, adds the results with the corresponding accumulator values, and then stores the final results in <dst>.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Explicit source operands cannot be accumulator.

This instruction doesn't support dword integers (D or UD).

14.2.24 mach – Multiply Accumulate High

Opcode	Instruction	Description
73 (0x49)	mach <dst> <src0> <src1>	Performing component-wise multiply accumulation of <src0>, <src1> and accumulator register, and returning the high dword of results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] mach[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mach[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mach[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      acc0.chan[n][63:0]
            = (src0.chan[n][31:16] *
      src1.chan[n][31:0])<<16
                  + acc0.chan[n][63:0];
      if (WrEn.channel[n] == 1) {
            dst.channel[n][31:0] = acc0.chan[n][63:32]
      }
}
```

Description:

The *mach* instruction performs dword integer multiply-accumulate operation and outputs the high dword (bits [63:32]). On a component by component basis, this instruction multiplies dwords in <src1> with the high words of dwords in <src0>, left-shifts the results by 16 bits, adds them with the corresponding accumulator values, and keeps the whole 64-bit results in the accumulator. It then stores the high dword (bits [63:32]) of the results in <dst>.

This instruction is intended to be used to emulate 32-bit dword integer multiplication by utilizing the large number of bits available in the accumulator. For example, the following three instructions perform vector multiplication of two

32-bit signed integer source from r2 and r3 and store the resulting vectors with high 32-bit in r4 and low 32-bit in r5.

```
mul   (8) acc0:d  r2.0<8;8,1>d   r3.0<8;8,1>:d
mach  (8) r4.0<1>:d  r2.0<8;8,1>d   r3.0<8;8,1>:d
mov   (8) r5.0<1>:d  acc0:d
```

As *mach* is used to generate part of 64-bit dword integer results, saturation modifier should not be used. In fact, saturation modifier should not be used for any of these three instructions.

Source and destination operands must be dword integers. Source and destination must be of the same type, signed integer or unsigned integer.

- If <dst> is UD, <src0> and <src1> may be UD and/or D. However, if any of <src0> and <src1> is D, source modifier, (abs), must be present to convert it to match with <dst>.
- If <dst> is D, <src0> and <src1> must also be D. They cannot be UD as it may cause unexpected overflow because the computed results are limited to 64 bits.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Therefore, explicit source operands cannot be accumulator.

14.2.25 mov – Move

Opcode	Instruction	Description
1 (0x01)	mov <dst> <src0>	Component-wise move from <src0> to <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] mov[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] mov[.<cmod>] (<exec_size>) reg reg
[(<pred>)] mov[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n]
    }
}
```

Description:

The *mov* instruction moves the components in <src0> into the channels of <dst>. If <src0> and <dst> are of different types, format conversion is performed. If <src0> is a scalar immediate, the immediate value is loaded into all channels of <dst>.

Restrictions:

The accumulator can be either <src0> or <dst> but not both.

This instruction does not implicitely update accumulator register.

14.2.26 mrest – Mask Restore

Opcode	Instruction	Description
45 (0x2D)	mrest <mstack> <mask>	Restoring <mask> from <mstack> and also updating <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format:

```
mrest (1) <mask> <mstack>
```

Syntax:

```
mrest (1) <mask> <mstack>
```

Pseudocode:

```
<mask> = <mstack>.TopOfStack;
<mstack>.pop(1);
```

Description:

The *mrest* instruction restores the value of a mask register from the specified mask-stack.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction restores <mask> from the top of stack of <mstack>, but it does not pop the mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.

14.2.27 msave – Mask Save

Opcode	Instruction	Description
44 (0x2C)	msave <mstack> <mask>	Pushing <mask> onto <mstack> and updating <mask> with the current EMask.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format:

```
msave (1) <mstack> <mask>
```

Syntax:

```
msave (1) <mstack> <mask>
```

Pseudocode:

```
<mstack>.push(<mask>);
<mask> = EMask;
```

Description:

The *msave* instruction pushes the value held in the specified mask register to the specified mask-stack, and then updates the mask value to the current execution mask (EMask). This is typically used just prior to entering a nested loop.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction updates <mask>, but does not update any mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.

14.2.28 mul – Multiply

Opcode	Instruction	Description
65 (0x41)	mul <dst> <src0> <src1>	Performing component-wise multiplication of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] mul[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mul[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mul[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] * src1.chan[n];
   }
}
```

Description:

The *mul* instruction performs component-wise multiplication of <src0> and <src1> and stores the results in <dst>.

When both <src0> and <src1> are of type D or UD, only the lower 16 bits of each element of <src0> are used. Accumulator maintains full 48-bit precision. Together with *mach* and *mov* instructions, full precision 64 bits multiplication results can be produced. (For that purpose, the type of <dst> must UD for the last mov instruction to get the lower dwords of partial results out.)

Multiplication of two floating point numbers follows rules in Table 14-5 (or Table 14-6), if the current floating point mode is IEEE mode (or ALT mode).

Table 14-5. Floating point multiplication of A (column) and B (row) in IEEE mode

	–inf	**–finite**	**–1.0**	**–denorm**	**–0**	**+0**	**+denorm**	**+1.0**	**+finite**	**+inf**	**NaN**
–inf	+inf	+inf	+inf	NaN	NaN	NaN	NaN	–inf	–inf	–inf	NaN
–finite	+inf	*	–A	+0	+0	–0	–0	A	**	–inf	NaN
–1.0	+inf	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–inf	NaN
–denorm	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
–0	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
+0	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+denorm	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+1.0	–inf	B	–1.0	–0	–0	+0	+0	+1.0	B	+inf	NaN
+finite	–inf	**	–A	–0	–0	+0	+0	A	*	+inf	NaN
+inf	–inf	–inf	–inf	NaN	NaN	NaN	NaN	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Note:

*	Result may be {+finite, +inf (overflow)}
**	Result may be {–inf (overflow), –finite}

Table 14-6. Floating point multiplication of A (column) and B (row) in ALT mode

	– fmax	**–finite**	**–1.0**	**–denorm**	**–0**	**+0**	**+denorm**	**+1.0**	**+finite**	**+fmax**	*******
– fmax	+fmax	+fmax	+fmax	NaN	NaN	NaN	NaN	–fmax	–fmax	–fmax	▨
–finite	+fmax	*	–A	+0	+0	–0	–0	A	**	–fmax	▨
–1.0	+fmax	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–fmax	▨
–denorm	+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	+0	+0	+0	+0	▨
+denorm	–0	–0	–0	–0	–0	+0	+0	+0	+0	+0	▨
+1.0	–fmax	B	–1.0	–0	–0	+0	+0	+1.0	B	+fmax	▨
+finite	–fmax	**	–A	–0	–0	+0	+0	A	*	+fmax	▨
+fmax	–fmax	–fmax	–fmax	–0	–0	+0	+0	+fmax	+fmax	+fmax	▨
*******	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

Note:

*	Result may be {+finite, +fmax (overflow)}
**	Result may be {–fmax (overflow), –finite}
***	Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Source operands cannot be an accumulator register.

When operating on integers with at least one of the source being a dword type (signed or unsigned), the destination cannot be a float (implementation note: the data converter only looks at the lower 34 bits of the results).

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

When operating on integers with at least one of the source being a dword type (signed or unsigned), the Overflow and Sign flags are undefined. Therefore, conditional modifier and instruction operation '.sat' cannot be used.

14.2.29 nop – No Operation

Opcode	Instruction	Description
126 (0x7E)	nop	Issuing an dummy instruction and performing no operation.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
nop
```

Syntax:

```
nop
```

Pseudocode:

```
n/a
```

Description:

The *nop* instruction takes an instruction dispatch but performs no operation. It may be used for assembly patching in memory, or be used to insert an instruction delay in the program sequence.

The *nop* instruction takes no operands, no instruction modifier, no conditional modifier and no predication.

14.2.30 not – Logic Not

Opcode	Instruction	Description
4 (0x04)	not <dst> <src0>	Performing component-wise logic NOT of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] not[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] not[.<cmod>] (<exec_size>) reg reg
[(<pred>)] not[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = !src0.chan[n]
   }
}
```

Description:

The *not* instruction performs logical NOT operation (or one's compliment) of <src0> and storing the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

This instruction does not work with float type operands.

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not implicitely update accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

14.2.31 or – Logic Or

Opcode	Instruction	Description
6 (0x06)	or <dst> <src0> <src1>	Performing component-wise logic OR of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] or[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] or[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] or[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] | src1.chan[n];
   }
}
```

Description:

The *or* instruction performs component-wise logic OR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.

This instruction does not implicitely update accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

14.2.32 pop – Mask Stack Pop

Opcode	Instruction	Description
47 (0x2F)	pop <mstack> <depth>	Popping <depth> level from <mstack> and updating <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format:

```
pop (1) <mstack> <depth>
```

Syntax:

```
pop (1) <mask> reg32
pop (1) <mask> imm32
```

Pseudocode:

```
<mstack>.pop(1);
```

Description:

The *pop* instruction pops 'n' levels off the specific mask-stack structure. The <depth> field specified the number of levels to discard from the stack, and is always treated as an unsigned integer value.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction does not update the mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.

As the mask stacks are 16 deep, <depth> must be an unsigned dword integer with bits [31:4] set to zero. Effectively, it is a 4-bit unsigned integer.

14.2.33 push – Mask Stack Push

Opcode	Instruction	Description
46 (0x2E)	push <mstack> <mask>	Pushing <mask> onto <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format:

```
push (1) <mstack> <mask>
```

Syntax:

```
push (1) <mstack> <mask>
```

Pseudocode:

```
<mstack>.push(<mask>);
```

Description:

The *push* instruction pushes the value held in the specified mask register to the specified mask-stack, and leaves the mask unchanged.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time. <exec_size> must be set to 1 (for example by Assembler).

If SPF is set, this instruction does not update the mask stack.

Restrictions:

Predication is not allowed.

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.

14.2.34 rndd – Round Down

Opcode	Instruction	Description
69 (0x45)	rndd <dst> <src0>	Taking component-wise floating point downward rounding of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] rndd[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndd[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndd[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = floor(src0.chan[n]);
   }
}
```

Description:

The *rndd* instruction takes component-wise floating point downward rounding (to the integral float number closer to negative infinity) of <src0> and storing the rounded integral float results in <dst>. This is commonly referred to as the *floor()* function.

This instruction only applies to floating point source operands. Destination may be of float type or of integer type.

This is the only single instruction floating point rounding operation. The other three rounding modes (up, to-even, to-zero) require two instructions.

Output data <dst> and conditional flag Increment (IN) for floating point rounding-down follow rules in Table 14-7 (or Table 14-8), if the current floating point mode is IEEE mode (or ALT mode). Note again that conditional flag IN is 0 for all cases.

Table 14-7. Floating point round-down in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	0	0	0	0	0	0	0	0
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								

Table 14-8. Floating point round-down in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	***
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	▨
IN	0	0	0	0	0	0	0	0	▨
Notes:									
*	Result may be {–finite, –0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if <src0> is {–inf, +inf, NaN}.								

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.

14.2.35 rndu – Round Up

Opcode	Instruction	Description
68 (0x44)	rndu <dst> <src0>	Taking component-wise floating point upward rounding of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] rndu[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndu[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndu[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = floor(src0.chan[n]);
      if (src0.chan[n]-floor(src0.chan[n]) > 0.0f)
         dst.increment[n] = 1;
      else
         dst.increment[n] = 0;
   }
}
```

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-up rounding increment stored in the increment bits. The rounding increments must be added to the results in <dst> to create the final round-up values to emulate the round-up operation, commonly known as the *ceiling()* function.

This instruction only applies to floating point source operands. Destination operand may be a floating point or an integer type operand.

The *ceiling()* function can be emulated in two instructions using rndu, for example, as

```
[(<pred>)] rndu.r.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

The first instruction stores the round up increments in a flag register, f#, by turning on the conditional modifier. The second instruction increments <dst> by 1.0 predicated by f#. This works even when predication is on for the *rndu* instruction, as the set bits in the resulting flag register, f#, is a subset of the enabled channels of the previous instruction. Output data <dst> and conditional flag Increment (IN) for floating point rounding-up follow rules in Table 14-9 (or Table 14-10), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN is 1 only if source data is a finite non-integral float.

Table 14-9. Floating point round-up in IEEE mode

<src0>	**–inf**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+inf**	**NaN**
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	***	0	0

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.

Table 14-10. Floating point round-up in ALT mode

<src0>	**– fmax**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+ fmax**	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	▨
IN	0	***	0	0	0	0	***	0	▨

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.
****	Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

This instruction cannot take accumulator as source or destination operand. This instruction does change the content of accumulator, however, the results in the accumulator are undefined.

14.2.36 rnde – Round to Even

Opcode	Instruction	Description
70 (0x46)	Rnde <dst> <src0>	Taking component-wise floating point round-to-even operations of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] rnde[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rnde[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rnde[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = floor(src0.chan[n]);
      if (src0.chan[n]-floor(src0.chan[n]) > 0.5f) {
         dst.increment[n] = 1;
      } else if (src0.chan[n]-floor(src0.chan[n]) < 0.5f) {
         dst.carry[n] = 0;
         } else {
            if (dst.chan[n] is odd) {
               dst.increment[n] = 1;
            } else {
               dst.increment[n] = 0;
         }
      }
   }
}
```

Description:

The *rnde* instruction takes component-wise floating point round-to-even operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in <dst> to create the final round-to-even values to emulate the round-to-even operation, commonly known as the *round()* function. The final results are the one of the two integral float values that is nearer to the input values. If the neither possibility is nearer, the even alternative is chosen.

This instruction only applies to floating point operands. Similar to *ceiling()*, the *round()* function can be emulated in two instructions using *rnde*, for example, as

```
[(<pred>)] rnde.c.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

Output data <dst> and conditional flag Increment (IN) for floating point rounding-to-even follow rules in Table 14-11 (or Table 14-12), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN may be 0 or 1 if source data is a finite non-integral float.

Table 14-11. Floating point round-to-even in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	***	0	0

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise.

Table 14-12. Floating point round-to-even in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	
IN	0	***	0	0	0	0	***	0	

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise.
****	Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.

14.2.37 rndz – Round to Zero

Opcode	Instruction	Description
71 (0x47)	rndz <dst> <src0>	Taking component-wise floating point round-to-zero operations of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT]	[FLT] [INT]

Format:

```
[(<pred>)] rndz[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndz[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndz[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = floor(src0.chan[n]);
      if (abs(src0.chann[n]) < abs(dst.chan[n])) {
         dst.increment[n] = 1;
      } else {
         dst.increment[n] = 0;
      }
   }
}
```

Description:

The *rndz* instruction takes component-wise floating point round-to-zero operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in <dst> to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the *truncate()* function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.

This instruction only applies to floating point operands. Similar to *ceiling() and round()*, the *truncate()* function can be emulated in two instructions using *rndz* instruction. For example, as

```
[(<pred>)] rndz.c.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

Output data <dst> and conditional flag Increment (IN) for floating point rounding-to-zero follow rules in Table 14-13 (or Table 14-14), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN is 0 for a source that is a positive float, and is 1 only for negative non-integral float.

Table 14-13. Floating point round-to-zero in IEEE mode

<src0>	**–inf**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+inf**	**NaN**
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	0	0	0

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.

Table 14-14. Floating point round-to-zero in ALT mode

<src0>	**– fmax**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+ fmax**	********
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	▨
IN	0	***	0	0	0	0	0	0	▨

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.
****	Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.

14.2.38 sad2 – Sum of Absolute Difference 2

Opcode	Instruction	Description
80 (0x50)	sad2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, and storing the scalar result to the first channel per 2-tuple in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
	•	•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] sad2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = abs(src0.chan[n] - src1.chan[n])
                  + abs(src0.chan[n+1] - src1.chan[n+1]);
}
```

Description:

The *sad2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1> and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision.

Destination register must have a stride of 2 bytes and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.

14.2.39 sada2 – Sum of Absolute Difference Accumulate 2

Opcode	Instruction	Description
81 (0x51)	sada2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, added to that from the accumulator, and storing the scalar result to the first channel per 2-tuple in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
	•	•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] sada2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] sada2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] sada2[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
      uwTmp = abs(src0.channel[n] - src1.channel[n])
            + abs(src0.channel[n+1] - src1.channel[n+1])
      if (WrEn.channel[n] == 1) {
            dst.channel[n] = uwTmp + acc[n]
      }
}
```

Description:

The *sada2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1>, adds the intermediate result with the accumulator value corresponding to the first channel, and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision. Higher precision (guide bits) stored in the accumulator allows multiple rounds (64 rounds) of sada2 instructions to be issued back to back without overflow the accumulator.

Destination register must have a stride of 2 bytes and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.

14.2.40 sel – Select

Opcode	Instruction	Description
2 (0x02)	(pred) sel <dst> <src0> <src1>	Component-wise selective move from <src0> or <src1> to <dst> based on predication.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•		•	[FLT] [INT]	[FLT] [INT]

Format:

```
(<pred>) sel (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
(<pred>) sel (<exec_size>) reg reg reg
(<pred>) sel (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn, NoPMask);
Evaluate(PMask);
for (n = 0; n < exec_size; n++) {
      if (r.channel[n] == 1) {
            if (PMask.channel[n] == 1) {
                  dst.channel[n] = src0.channel[n]
            } else {
                  dst.channel[n] = src1.channel[n]
            }
      }
}
```

Description:

The *sel* instruction selectively moves the components in <src0> or <src1> into the channels of <dst> based on the predication. On a channel by channel basis, if the channel condition is true, data in <src0> is moved into <dst>; Otherwise, data in <src1> is moved into <dst>.

As the predication is used to select the two sources, it is not included in the evaluation of WrEn. <pred> is mandatory. If it is <omitted>, the results are unpredictable.

If <src0>, <src1> and <dst> are of different types, format conversion is performed.

Restrictions:

Destination channels cannot be on odd-byte sub-register locations. In other words, when destination is of byte type, destination horizontal stride cannot be 1. If destination horizontal stride is not 1, destination register region origin cannot be on an odd byte location. This is because that the conditional flag for execution channels that have minimal granularity of word are used by this instruction.

This instruction does not implicitly update accumulator registers.

14.2.41 send – Send Message

Opcode	Instruction	Description
49 (0x31)	send <post_dest> <curr_dest> <src0> <desc>	Performing an implied move from <src0> to <curr_dest>, sending a message stored in MRF starting at <curr_dest> to a shared function identified by <desc> with a GRF writeback location at <post_dest>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] send (<exec_size>) <pdst> <cdst> <src0> <desc>
```

Syntax:

```
[(<pred>)] send (<exec_size>) reg reg reg reg32a
[(<pred>)] send (<exec_size>) reg reg reg imm32
```

Pseudocode:

```
Evaluate(EMask);
if (src0 != null) {
   MRFREG <mdest> = <curr_dest>.RegNum with
<post_dest>.region;
      for (n = 0; n < exec_size; n++) {
      mdest.chan[n] = src0.chan[n];
      }
}
if (post_dest != null) {
      <ResponseReg> = <post_dest>.RegNum;
}
<MsgChEnable> = EMask;
<SourceReg> = <curr_dest>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>,
<desc>;
```

Description:

The *send* instruction performs data communication between a thread and external function units, including shared functions (Extended Math unit, Sampler, Data Port Read, Data Port Write and URB) and some fixed functions (e.g. Thread Spawner, who also have an unique Shared Function ID). The *send* instruction adds an entry to the EU's message request queue. The request message is stored in a block of contiguous MRF registers. The response message, if present, will be returned to a block of contiguous GRF registers. The return GRF writes may be in any order depending on the external function units. The current destination operand, <curr_dest>, is the lead MRF register for request. The posted destination operand, <post_dest>, is the lead GRF register for response. The message descriptor field <desc> contains the Message Length (the number of consecutive MRF registers) and the Response Length (the number of consecutive GRF registers). It also contains the target function unit ID and function control signals. The bit field definition of <desc> can be found in Table 14-15. EMask is forwarded to the target function in the message sideband.

In addition to enqueueing message request, the *send* instruction also performs an implied move from source operand <src0> to the current MRF destination <curr_dest>. The implied move is not subject to EMask. The *send* instruction is the only way to terminate a thread. When the EOT (End of Thread) bit of <desc> is set, it indicates the end of thread to the EU, the Thread Dispatcher and, in most cases, the parent fixed function.

Message descriptor field <desc> can be a 32-bit immediate, *imm32*, or a 32-bit scalar register, <reg32a>. The MSB of the message descriptor, the EOT field, always comes from bit 127 of the instruction word, which is the MSB of imm32. GEN4 restricts that the 32-bit scalar register <reg32a> must be the leading dword of the address register. It should be in the form of a0.0<0;1,0>:ud. When <desc> is a register operand, only the lower 31 bits of <reg32a> are used. A thread must terminate with a *send* instruction with EOT turned on.

<curr_dest> is a 256-bit aligned physical MRF register. Only the register number is present in the instruction word. It serves for two purposes: as the leading MRF register of the request message and as the destination of the implied move.

<src0> is used to signal whether there is an implied move or not. It can be either a null register or a GRF register. For other cases, hardware behavior is undefined. <src0> cannot be any ARF and cannot be register-indirect-addressed, as address register may be used as the operand for <desc>. If <src0> is a null register, there is no implied move. If it is a GRF register, there is an implied move from GRF to MRF. The implied move is fully described based on the parameters of the 'send' instruction. The destination operand of the implied move, <move_dest>, assumes the register number of <curr_dest> and the rest parameters from <post_dest>. The source operand of the implied move is <src0>. The implied move also assumes the same <exec_size>. The implied move turns on all the write enables for MRF destination, therefore, the whole MRF register of <curr_dest> is updated regardless of the content of EMask. The subfields belonging to the region of <move_dest> contain valid data, other subfields are undefined. The implied move may be used to save an instruction of moving the message header from GRF to MRF. It may also be used for header-less message going to the Extended Math unit by taking advantage of source swizzle specifications. Specifically, the 'send' instruction with implied move makes the

computation on Extended Math unit just like other arithmetic operations performed by the EU. In that case, the full MRF register can be viewed as pipeline storage to hold intermediate data post source swizzle and before the Extended Math unit performs the actual computation.

<post_dest> serves for three purposes: to provide the leading GRF register location for the response message if present, to provide parameters to form the channel enable sideband signals, and to provide the subregister parameters for the implied move.

<post_dest> signals whether there is a response to the message request. It can be either a null register, a direct-addressed GRF register or a register-indirect GRF register. Otherwise, hardware behavior is undefined.

If <post_dest> is null, there is no response to the request. Meanwhile, the Response Length field in <desc> must be 0. Certain types of message requests, such as memory write (store) through the Data Port, do not want response data from the function unit. If so, the posted destination operand can be null.

If <post_dest> is a GRF register, the register number is forwarded to the shared function. In this case, the target function unit must send one or more response message phases back to the requesting thread. The number of response message phases must match the Response Length field in <desc>, which of course cannot be zero. For some cases, it could be an empty return message. An empty return message is defined as a single phase message with all channel enables turned off.

The subregister number, horizontal stride, destination mask and type fields of <post_dest> are always valid and are used in part to generate the EMask. This is true even if <post_dest> is a null register (this is an exception for null as for most cases these fields are ignored by hardware). These parameters of <post_dest> follow the same restriction as that of normal destination operand – destination region cannot cross the 256-bit register boundary.

If the implied move is performed, the subregister number, horizontal stride and type fields of <post_dest> are used to describe the implied MRF destination register region.

The EMask corresponds to the 256-bit aligned data fields of the type of <post_dest>. EMask can have up to 16 bits. Therefore, fundamental type of <post_dest> can be either word or dword, but not byte. If <post_dest> is of dword, lower 8 bits of EMask (as dword enables) are generated based on the instruction parameters, and the higher 8 bits are forced to zero by hardware. If <post_dest> is of word, all 16 bits are generated. In other words, if execution size is 16, <post_dest> must be 256-bit aligned and must be of fundamental type of word. If execution size is 8 or less (yes, smaller execution size is allowed), the fundamental type of <post_dest> can be word or dword. The generation of EMask is subject to the following parameters: predication control if present, execution size, {subregister number, mask if present, horizontal stride if present, type} of <post_dest>, composite mask (from A/L/I/Cmask) if NoMask is not present, SecHalf if present. Only the 16-bit EMask is forwarded to the target function as the channel enable sideband signals; none of the input parameters are sent out. Therefore, these parameters may be manipulated to form a desired EMask according to the specifications of the target function. For example, in order to forward the 16-bit dword enables to the Data Port Write unit for a scattered dword store message, <post_dest> may be set to a null register of 'uw' type, even though the actual data stored in MRF are floating point values.

The 16-bit channel enables of the message sideband are formed based on the above mentioned EMask. Interpretation of the channel enable sideband signals is subject to the target external function. In general for a 'send' instruction with return messages, they are used as the destination dword write mask for the GRF registers starting at <post_dest>. For a message that has multiple return phases, the same set of channel enable signals applies to all the return phases.

Instruction compression is not allowed for this instruction. The hardware behavior is undefined if this instruction is set as compressed. However, compress control can be set to "SecHalf" to affect the EMask generation.

Thread managed memory coherency: A special usage of using non-null <post_dest> is to support write-commit signaling for memory write service by the Data Port Write unit. If <post_dest> is not null for a memory write request, the Data Port along with the Data Cache or Render Cache will wait until all the posted writes for the request have reached the coherent domain before sending back to the requesting thread an empty message to <post_dest> register. A memory write reaching the coherent domain, also referred to as reaching the global observable state, means that subsequent read to the same memory location, no matter which thread issues the read, must return the data of the write.

The destination dependency control, {NoDDClr}, can be used in this instruction. This allows software to control the destination dependencies for multiple 'read'-type messages similar to that for multiple instructions using EU execution pipeline. As *send* does not check register dependencies for the post destination, {NoDDChk} should not be used for this instruction.

Table 14-15. Message Descriptor Definition

Bit	Description
31	**End Of Thread** This field, if set, indicates that this is the final message of the thread and the thread's resources can be reclaimed. This field is also valid when the Message Descriptor is from a register. However, in that case, this field comes from the instruction word instead of the register.
30:28	Reserved : MBZ
27:24	**Target Function ID** This field indicates the function unit for which the message is intended. 0000 = Null 0001 = Extended Math 0010 = Sampling Engine 0011 = Message Gateway 0100 = Data Port Read 0101 = Data Port Write 0110 = URB 0111 = Thread Spawner 1000--1111: Reserved

23:20	**Message Length.** This field specifies the number of 256-bit MRF registers starting from <curr_dest> to be sent out on the request message payload. Valid value ranges from 1 to 15. A value of 0 is considered erroneous.
	Format = U4
	Range = [1,15]

19:16	**Response Length.** This field indicates the number of 256-bit registers expected in the message response. The valid value ranges from 0 to 8. A value 0 indicates that the request message does not expect any response. The largest response supported by GEN4 is 8 GRF registers.
	Format = U4
	Range = [0,8]

15:0	**Function Control** This field is intended to control the target function unit. Refer to the section on the specific target function unit for details on the contents of this field.

Table 14-16 provides a summary of the signals associated with each message that is sent to the shared function. It contains fields from the send instruction as well as fields from control register *cr0* and state register *sr0*.

Table 14-16. Sideband Signals Associated with Each Message Sent to the Shared Function

Signal	Bits	Source
EOT	1	End of Thread: Sourced from the EOT bit in ***send*** instruction word
SFID	3	Shared Function Identifier: Sourced from the target function ID field in <desc> of ***send***
MLEN	4	Message Length: Sourced from the message length field in <desc> of ***send***
RLEN	4	Response Length: Sourced from the response length field in <desc> of ***send***
FC	16	Function Control: Sourced from the function control field in <desc> of ***send***
REG	7	Destination Register: Sourced from the 256-bit register aligned register number of the <post_dest> field of ***send***
CE	16	Channel Enable: Sourced from the execution mask of ***send***
CLEAR	1	Destination Register Clear: Source from the Destination Dependency Control field (inverse of NoDDClr) in ***send*** instruction word
FFID	4	Fixed Function Identifier: Sourced from the Fixed Function ID field in *sr0*
EUID	4	Execution Unit Identifier: Sourced from the EUID field in *sr0*
TID	2	Thread Identifier: Sourced from the TID field in *sr0*
FPMODE	1	Floating Point Mode: Sourced from the floating point mode field in *cr0*

Restrictions:

Software must obey the following rules in signaling the end of thread using the *send* instruction:

- The posted destination operand must be null.
  - No acknowledgement is allowed for the *send* instruction that signifies the end of thread. This is to avoid deadlock as the EU is expecting to free up the terminated thread's resource.
- A thread must terminate with a *send* instruction with message to a shared function on the output message bus; therefore, it cannot terminate with a *send* instruction with message to the following shared functions: Sampler unit, Extended Math unit, NULL function
  - For example, a thread may terminate with a URB write message or a render cache write message.
- A root thread originated from the media (generic) pipeline must terminate with a *send* instruction with message to the Thread Spawner unit. A child thread should also terminate with a send to TS. Please refer to the Media Chapter for more detailed description.

This instruction does not implicitly update accumulator registers.

Implementation Restrictions: As the hardware does not check for post destination dependencies on this instruction, software must ensure that there is no destination hazard for the case of 'write followed by a posted write' shown in the following example.

1. mov r3 0
2. send r3.xy <rest of send instruction>
3. mov r2 r3

Due to no post-destination dependency check on the 'send', the above code sequence could have two instructions (1 and 2) in flight at the same time that both consider 'r3' as the target of their final writes. The first write to complete will release the dependency on 'r3' for subsequent instructions, thus allowing instruction 3 to proceed, even though one of the two prior instructions is still in-flight and will eventually modify 'r3'. Thus the proper instruction-order contents of 'r3' cannot be guaranteed for such a code sequence, and such sequence should not be used.

A proper code sequence which does guarantee program-order writes to 'r3' in this example is shown below. The solution is to use {NoDDClr} to the first instruction. Due to in-order instruction dispatch and known/fixed execution pipeline latencies, the posted write from the second instruction cannot bypass the first instruction, r3 update order from instructions 1 and 2 are ensured. So any dependencies on 'r3' will only be cleared by the posted write from the second instruction. Therefore, the third instruction will access the correct data from r3.

1. mov r3 0 {NoDDClr}
2. send r3.xy <rest of send instruction>
3. mov r2 r3

Another coding option is to insert a dummy move before the send instruction to stall the thread's execution until the first write to 'r3' has retired, this avoiding the potential destination hazard. However, this approach will consume additional instruction cycles waiting for 'r3' to retire.

1. mov r3 0
2. mov null r3 // This instruction clears the scoreboard for r3
3. send r3.xy <rest of send instruction>
4. mov r2 r3

Yet another coding option is to use {**switch**} instruction option on the instruction preceding the send instruction. The 'switch' instruction option will flush the execution pipe, ensuring all writes in the execution pipe to be retired.

1. mov r3 0
2. <any instruction> {switch}// This instruction flushes the execution pipe
3. send r3.xy <rest of send instruction>
4. mov r2 r3

This coding option is particularly useful for send instruction with indirect-addressed post destination, where the post destination registers are not known to the programmer.

As the retirement order for post-destinations of different send instructions may be unknown (e.g. from different shared functions), a posted write overlapping with a previously-issued posted write should be avoided in general, particularly for send instructions with indirect post-destinations.

[DevBW, DevCL] Errata: A destination register from a send cannot be used as a destination register until after it has been sourced by an instruction with a different destination register.

14.2.42 shl – Shift Left

Opcode	Instruction	Description
9 (0x09)	shl <dst> <src0> <src1>	Performing component-wise logic left shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] shl[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] << src1.chan[n]
   }
}
```

Description:

The *shl* instruction performs component-wise logical left shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1>, where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored. The results are NOT stored in the accumulator register.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel. <src0> and <dst> can be signed or unsigned integers and can be of different types. This instruction does not work with float type operands. Saturation modifier is only allowed when this instruction is in packed-word mode. Hardware detects overflow properly and use it to perform saturation operation on the output, as long as the shifted result is within 33 bits. Otherwise, the result is undefined.

Results of saturation in packed-dword mode are unpredicable.

Restrictions:

This instruction does not work with float type operands.

This instruction does not implicitely update accumulator register.

14.2.43 shr – Shift Right

Opcode	Instruction	Description
8 (0x08)	shr <dst> <src0> <src1>	Performing component-wise logic right shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] shr[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] shr[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] shr[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] >> src1.chan[n]
   }
}
```

Description:

The *shr* instruction performs component-wise logical right shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1> where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction only takes on unsigned sources. When <src0> contains unsigned integers, no source modifier is allowed. <src0> is only allowed to be signed integer if source modifier (abs) is used. *Note: for unsigned sources, the behavior of shr and asr are effectively the same.*

Restrictions:

This instruction does not work with float type operands.

This instruction does not implicitely update accumulator register.

14.2.44 wait – Wait Notification

Opcode	Instruction	Description
48 (0x30)	wait <nreg>	Waiting for notification on the notification register <nreg>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
wait (<exec_size>) <nreg>
```

Syntax:

```
wait (1) n#
```

Pseudocode:

```
n/a
```

Description:

The *wait* instruction evaluates the value of the notification count register <nreg>. If <nreg> is zero, the execution of the thread is stalled and the thread is put in 'wait_for_notification' state. If <nreg> is not zero (i.e., one or more notifications have been received), <nreg> is decremented by one and the thread continues executing on the next instruction. If a thread is in the 'wait_for_notification' state, when a notification arrives, the notification count register is incremented by one. As the notification count register becomes non-zero, the thread wakes up to continue execution and at the same time the notification register is decremented by one. If there was only one notification arrived, the notification register value becomes zero. However, during the above mentioned time period, it is possible that more notifications may arrive, making the notification register non-zero again.

When multiple notifications are received, software must use 'wait' instruction to decrement notification count register for each of the notifications.

Notification register n0:ud is for thread to thread communication (through message gateway shared function) and n1:ud for host to thread communication (through MMIO registers). See Message Gateway chapter for thread-thread communication and Debug chapter for host-to-thread communication.

Restrictions:

Only one source operand.

<src0> and <dst> must be n0 or n1, <src1> must be *null*.

Execution size must be 1 as the notification registers are scalar.

Predication is not allowed.

Implementation restriction: Two back-to-back *wait* instructions in a program (without any instruction in between) are not allowed. As a minimal, a *nop* has to be inserted between two *wait* instructions.

14.2.45 while – While

Opcode	Instruction	Description
39 (0x27)	while <exitcode>	Marking the end of a do-while block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] while (<exec_size>) <exitcode>
```

Syntax:

```
[(<pred>)] while (<exec_size>) imm32
```

Pseudocode:

```
Evaluate(EMask, NoCMask);
if (EMask != 0) {
   CMask = EMask;
   LMask = EMask;
   Jump(<exitcode.IPCount>);
} else {
   CMask = LStack.TopOfStack;
   LStack.Pop(1);
   LMask = LStack.TopOfStack;
   LStack.Pop(1);
}
```

Description:

The *while* instruction marks the end of a do-while block. The instruction first evaluates the loop termination condition for each channel as determined by the bit-wise AND of the AMask, LMask, and the predication flag specified in the instruction. Both CMask and LMask are updated to this value for any further passes through the loop. If any channel has not terminated as indicated by its bit position = 1, a branch is taken to a destination address based specified in the instruction, and the loop continued for those channels.

Once all channels have been suspended (as indicated by LMask = ...000b), the loop is terminated by popping CMask and then LMask from the LStack, enabling those data channels active prior to entering the loop. The instruction sequence then falls through and execution continues with the instruction immediately following the 'while'. Note that this instruction also updates CMask to ensure that CMask is always a subset of LMask.

Also note that the EMask evaluation does not consider CMask as the channels temporally turned off by CMask in the current loop need to be considered in the conditional evaluation in the *while* instruction.

The following table describes the 32-bit exit code <exitcode>. <InstCount> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction after the *do* instruction of the do-while block of code. It should be a negative number for the backward referencing.

In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:16	Reserved: MBZ
15:0	**InstCount (Jump Instruction Count).** This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction pops both CMask and LMask from LStack upon falling through.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When this instruction falls through, it restores the CMask and LMask from LStack.TopOfStack, which supposes to be static for code with single program flow.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

14.2.46 xor – Logic Xor

Opcode	Instruction	Description
7 (0x07)	xor <dst> <src0> <src1>	Performing component-wise logic XOR of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] xor[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] xor[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] xor[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
   if (WrEn.chan[n] == 1) {
      dst.chan[n] = src0.chan[n] ^ src1.chan[n];
   }
}
```

Description:

The *xor* instruction performs component-wise logic XOR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.

The results are NOT stored in the accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

15 EU Programming Guide

15.1 Assembler Pragmas

15.1.1 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*

15.1.2 Defaults and Defines

The default execution size is set according to the destination register type as the following

Destination Register Type	Default Execution Size
UB | B	(16)
UW | W	(16)
F | UD | D	(8)

The default execution size can be overwritten globally for all instructions using

.default_execution_size *(Execution_Size)*

or be set according the **destination** register type using

.default_execution_size_*Type* *(Execution_Size)*

The default register type can be set for all register files using

.default_register_type *Type*

or be set per register file using

.default_register_type_*RegFile* *Type*

The default **source** register region for all symbols can be set using

.default_source_register_region *<VirtStride; Width, HorzStride>*

or be set per register type using

.default_source_register_region_*type* *<VirtStride; Width, HorzStride>*

The default **destination** register region for all symbols can be set using

.default_destination_register_region *< HorzStride>*

or be set per register type using

.default_destination_register_region_*type* *< HorzStride>*

Finally, the precompiler supports the string replacement statement of .define in the following form

.define <symbol> Expression

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.

15.1.3 Example Pragma Usages

Example 1: Declaration for 8x4=32-Byte Regions:

The following symbol *Block* can be used to address any 8x4 byte region within the Cartisian system of a 16x8 byte GRF register area starting from r0.

Declaration	`// 32x4 Byte Array`
	`.declare Block Base=r0 ElementSize=1 Region=<32;8,1> Type=b`

Fully-Expressed Instr	`mov(32)    ?:b     r0.16<32;8,1>:b        //               r0`
	`xxxxxxxxxxxxxxxxoooooooo xxxxxxxx`
	`//  r1`
	`xxxxxxxxxxxxxxxxooooooooxxxxxxxx`
	`//  r2`
	`xxxxxxxxxxxxxxxxooooooooxxxxxxxx`
	`//  r3`
	`xxxxxxxxxxxxxxxxooooooooxxxxxxxx`

Short-handed Instr	`Mov     ?:b     Block(0,16)     //   (0,16):   RegNum=0, SubRegNum=16`

Example 2: Declaration for 8x1 Float Regions:
The following symbol *Trans* can be used to address any 8x1 float region within the Cartisian system of a 8x4 float GRF register area starting from r5.

Declaration	`// 8x4 float Array starting at r5`
	`.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`

Fully-Expressed Instr	`mov(8)     ?:f     r6.0<0;8,1>:f      //   2nd  16x1  Row  of Trans. Matrix`
	`// r5 FFFFFFFF`
	`// r6 00000000`
	`// r7 FFFFFFFF`
	`// r8 FFFFFFFF`

Short-handed Instr	`mov ?:f Trans(1) // RegNum = 5+1 = 6`

Example 3: 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 Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`
Fully-Expressed Instr	`mov(8) ?:f r[a0.0,224]<0;8,1>:f`
Short-handed Instr	`mov ?:f Trans[a0.0,2] // [a0.0 + 5*32 + 2*32]`

Example 4: Declaration with VxH Indirect Addressing:
The VxH register-indirect-register-addressing for Trans can be provided in the following short-hand form.

Declaration	`//8x4 float data array and word indices` `.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`
Fully-Expressed Instr	`mov(8) ?:f r[a0.0,224]<1,0>:f`
Short-handed Instr	`mov ?:f Trans[a0.0,2]<1,0> // [a0.0+224] [a0.1+224] … [a0.7+224]`

Example 5: Declaration with Vx1 Indirect Addressing:
As width (4) is smaller than the execution region size (8), multiple indexed registers are used.

Declaration	`//8x4 float data array and word address array` `.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`
Fully-Expressed Instr	`mov(8) ?:f r[a0.0,244]<4,1>:f`
Short-handed Instr	`mov ?:f Trans[a0.0,2]<4,1> //[a0.0+224] [a0.1+224]`

15.1.4 Assembly Programming Guideline

The following program skeleton illustrates the basic structure of a typical assembly program.

```
//  single line comment

/*
       block comment
*/
```

```
<preproc_directive>   // macros, include, etc.  Are global – handled by the pre-
processor
<preproc_directive>   // applies to all code that follows in sequence

// ------------ some kernel ---------------------------
.kernel <kernel_name_string>  // [REQUIRED]
                              // ------- Register requirements --------
 .reg_count_total     <uint>  // [REQUIRED] a more direct way to specify the exact
parameters require
 .reg_count_payload   <uint>  // [REQUIRED] rather than to have to indirectly do
that by adding the
                              //   the payload and temps together to get the total
(as is the case now)
                              // Note: no more "reg-count-temp"

                              // -------------- Defaults ---------------
 <default…>                   //  these  should  be  specified  per-kernel  and  have
only kernel-scope
 <default…>                   // Same defaults as those already defined in the ISA
doc, but just
 <default…>                   // moved  within  the  kernel  to  make  each  kernel
completely self-sufficient
                              // and not impacted defaults of earlier kernels

                              // --------- Memory Requirements ---------
                              // [optional] memory block info (just a placeholder
for now...)
 <MBDa>                       //     memory block descriptor a (TBD)
 <MBDb>                       //     memory block descriptor b (TBD)
 <MBDc>                       //     memory block descriptor c (TBD)
 <MBDd>                       //     memory block descriptor d (TBD)

                              // ---------------- Code  ----------------
 .code                        // [REQUIRED]
       <instruction>
       <instruction>
       <instruction>
    <LabelLine>               // labels are code-block scope
       <instruction>
       <instruction>

 .end_code                    // [REQUIRED]

.end_kernel                   // [REQUIRED]

// --------- next kernel -------------

// --------- next kernel -------------

// ...
```

15.2 Usage Examples

15.2.1 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 in 2's complement form. Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following figure.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
V7	V6	V5	V4	V3	V2	V1	V0

15.2.1.1 Supporting Pixel Shader Indexing

When a pixel shader program is converted to run on Gen4 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.

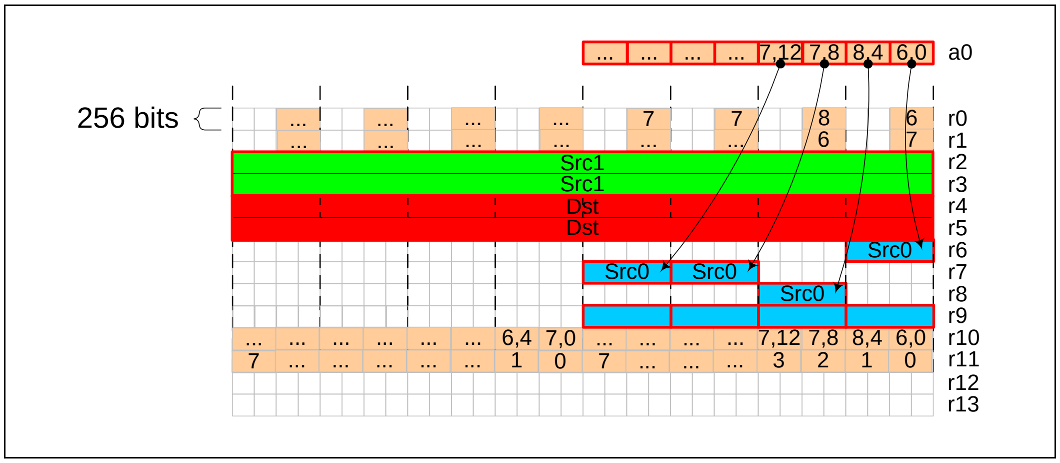
Consider a generic pixel shader instruction in the form of

```
op  r4  r[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 Gen4 instructions. The corresponding operations are illustrated in Figure 15-1.

```
mov (16)   r11.0<1>:w    0x01234567:v      // assigning a ramp vector, repeated
once

mul (16)    acc0:w      r11.0<0;16,1>:w   4:w// expand ramp range to 4 bytes per
step

mac (16)   r10.0<1>:w    r0.0<16;8,2>:w    32:w    // r10 = index*32 +
0|4|…|28|0|4…|28

mov (8)     a0.0<1>:w r10.0<0;8,1>:w

op (8)  r4.0<1>:f  r[a0.0]<1,0>:f      r2.0<0;8,1>:w // Operate on the first half

mov (8)     a0.0<1>:w r10.8<0;8,1>:w               // Index values are off by a reg
(32b)

op (8)  r5.0<1>:f  r[a0.0+32]<1,0>:f r3.0<0;8,1>:w // Operate on the second half.
```

Figure 15-1. 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.

15.2.1.2 Supporting OpenGL Vertex Shader Instruction SWZ

When an OpenGL Vertex Shader program is converted to run on Gen4 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 Gen4 native instruction. The vector immediate can significantly reduce the overhead of emulating such OpenGL instruction.

Consider an OpenGL Shader instruction in the form of

SWZ   r1   r0.0-zx-1   // Expected results: r1.x = 0; r1.y = -r0.z; r1.z = r0.x; r1.w = -1

It can be emulated by the following three Gen4 instructions.

mul(8) r1.0<1>:f   r0.xzxz   0x1F111F11:v // Constant vector of (1 -1 1 1 1 -1 1 1)

mov (1)   f0.0   8b’10011001   // Set flag & masked out channels y and z

(f0.0)mov(8) r1.0<1>:f   0x000F000F:v   // Constant vector of (0 0 0 -1 0 0 0 -1)

In case that only 0, 1, -1 channel replacement is used and there is no signed swizzle, it may be emulated in two Gen4 instructions. This is illustrated by the following example:

OpenGL:

SWZ   r1   r0.0zx-1   // Expected results: r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1

Gen4:

```
mov (1)      f0.08b'01100110                // Set flag and masked out channels x and w

(f0.0)sel (8) r1.0<1>:f r0.yzxy 0x000F000F:v  // Constant vector of (0 0 0 -1 0 0 0 -
1)
```

15.2.2 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 vertex shader of matrix-vector product that is implemented on Gen4 in the pair of 4-component vector mode. The 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 Gen4 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, NoDDChk}

dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f  {NoDDClr, NoDDChk}

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 Gen4 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
```

15.2.3 Null Register as the Destination

Null register can be used as the destination for most of the instructions. Here are some example usages.

- Null as destination for regular ALU instructions: As all ALU instructions can be configured to update the flag registers using the conditional modifiers, it is not necessary to have a destination register if the programmer only cares about the conditionals of the operation. In that case, a null in the destination operand field saves register space as well as one less dependency checking.
- Null as the destination for SEND/STOR instructions: for the send instruction that only send messages out to an external unit and does not require any return data or feedback, a null in the destination register field signifies the case.
  - One extension of such case is that even though the operation does not have any return values, a return phase with no payload but simply updating the scoreboard flag for a non-null register can provide a signaling mechanism between the thread and the target external unit. One application of this usage is to allow software to manage the coherency of shared memory resources such like the many caches in the system (particularly, valuable for read/write caches). This is not currently the POR for Gen4 though.

15.2.4 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 a 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 15-1. 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 15-2. 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.#
```

15.2.5 Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of shader instructions.

15.2.5.1 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 the following form:

```
[([!]p0.{select|any|all})] cos[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

For a SIMD4x2 VS implementation with the following register mappings:

```
    p0      →  f0.0
    src0    →  r0
    dest    →  r1
```

The equivalent Gen4 instruction is as the following:

```
[([!]f0.0.{select|any4h|all4h})] SEND (8) r1[.mask]:f m0 [-][(abs)]r0[.swizzle]:f
MATHBOX|COS[|SAT]
```

If the source swizzle is replication, the message description field can be modified to MATHBOX|COS|SCALAR to take advantage of the fast mode (scalar mode) supported by the Extended Math. The implied move of the SEND instruction is equivalent to the following instruction:

```
    MOV (8) m0[.mask]:f [-][(abs)]r0.0[.swizzle]:f {NoMask}
```

For a SIMD16 PS implementation, the register mappings are as the followings

```
    p0      →  f0…f3                        // in order of R, G, B, A
    src0    →  r0,r1; r2,r3; r4,r5; r6,r7
    dest    →  r8,r9; r10,r11; r12,r13; r14,r15
```

There are several ways to translate the instruction, depending on the operand/instruction modifiers present in the instruction. If predicate is not present and the source swizzle is replication, say, src0.y, which is r2-r3, the translation could be as the following instructions:

```
send (8) r8:f m0    -(abs)r2:f MATHBOX|COS
```

send (8) r9:f m1   -(abs)r3:f MATHBOX|COS {SecHalf}   // use the second half of 8 flag bits

mov (16) r10:f r8:f   // All destination color chan's are same

mov (16) r12:f r8:f   // MOV is faster than most MathBox func's

mov (16) r14:f r8:f   // These MOV's are compressed instructions

Notice that instead of issuing Extended Math messages with the same input data, destination color channel replication is performed by the MOV instructions. This is faster for the thread for most cases as many Extended Math functions consume multiple cycles. This also conserves message bus bandwidth as well as the usage of the shared resource – Extended Math. The destination mask in the instruction indicates which of the r8 to r15 registers are updated. If the source swizzle is not replication, there will be 8 SEND instructions.

With predication on, if the predication modifier is p0.select, translation is to take the selected flag register f#. The other predication modifiers '.any' and '.all' are translated into '.any4v' and '.all4v', respectively. Notice that with predication on, it is not required to run all 4 pixels in a subspan in the same way, so no need to enforce .any4h/.any4v. The following example shows the instruction with predication (but without .select modifier).

(f0[.any4v|.all4v]) send (8) r8:f m0   -(abs)r2:f MATHBOX|COS

(f0[.any4v|.all4v]) send (8) r9:f m1   -(abs)r3:f MATHBOX|COS {SecHalf}

(f1[.any4v|.all4v]) mov (16) r10:f r8:f   // All destination color chan's are same

(f2[.any4v|.all4v]) mov (16) r12:f r8:f   // MOV is faster than most MathBox func's

(f3[.any4v|.all4v]) mov (16) r14:f r8:f   // These MOV's are compressed instructions

The same instructions works also for predication with select component modifier. We simply replase f0 to f3 above by the selected flag register, say, f1. The modifier of any4h/all4v would also work.

15.2.5.2 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 that uses scratch memory as a source operand

[([!]p0.{select|any|all})] add dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

For a SIMD4x2 VS implementation with the following register mappings

p0	→	f0
src0	→	r0
src1	→	s2 / r10
dest	→	r1

In this example, the scratch memory offset is provided by an immediate and a GRF register r10 is used as the intermediate GRF location for spill/fill of scratch buffer accesses. This arithmetic instruction is converted into a Data Port read followed by an arithmetic instruction.

mov (8) r3:d r0:d {NoMask}   // move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}// s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask}   // s2 for vertex 1

send (8) r10 m0 r3 DATAPORT|RC|READ_SIMD2

[([!]f0.{sel|any4h|all4h})] add (8) r1[.mask]:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r10[.swizzle]:f

So if scratch register is the source, there is no need to use the channel enable side band. This is also true for channel-serial PS cases.

Now, let’s consider the case when a scratch register is the destination of an instruction.

p0	→	f0
src0	→	r0
src1	→	r1
dest	→	s2 / r10

We have

add (8) m1:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r1[.swizzle]:f mov (8) r3:d r0:d {NoMask}   // move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}// s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask}   // s2 for vertex 1

    [([!]f0.{sel|any4h|all4h})] send (8) **null**[.mask] m0 r3 DATAPORT|RC|WRITE_SIMD2

Notice that with a null as the posted destination register, we are able to transfer the [.mask] over the message channel enables. In many cases for scratch memory assess, a write-with-commit is required, therefore, the posted destination register could be r10.

Now, let's consider the PS case when a scratch register is the destination of an instruction.

p0	→	f0-f4
src0	→	r0-r7
src1	→	r8-r15
dest	→	s16-s23 / r16-r23

When predication is not on (or predication with swizzle control on), we have

add (16) m4:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m6:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m8:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m10:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (8) r3:d 0x76543210:v {NoMask}   // ramp function

mul (16) acc0:d r3:d 16 {NoMask}   // ramp function

add (8) acc0:d acc0:d 64 {NoMask,SecHalf}   // ramp function

add (16) m2:d acc0:d 2*256 {NoMask}   // ramp function

    send (16) **null** m1 r3 DATAPORT|RC|WRITE_SIMD16

As there is no bit left from the unit specified descriptor field, the 4-bit mask must be put into the header field in m1, which requires at least two more instructions.

Alternatively, or for the case that predication without modifier is on, we can do a read-modify-write.

mov (8) r3:d 0x76543210:v {NoMask}   // ramp function

mul (16) acc0:d r3:d 16 {NoMask}   // ramp function

```
add (8)  acc0:d acc0:d 64 {NoMask,SecHalf}    // ramp function

add (16) m2:d acc0:d 2*256 {NoMask}           // ramp function
    send (16) r16 m1 r3 DATAPORT|RC|READ_SIMD16       // read from scratch

// some of the following four instructions may be omitted based on [.mask] field

[([!]f0.{sel|any4v|all4v})] add (16) r16:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-
][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r18:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-
][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r20:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-
][(abs)] r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r22:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-
][(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (16) m4:f r16:f {NoMask}

mov (16) m6:f r18:f {NoMask}

mov (16) m8:f r20:f {NoMask}

mov (16) m10:f r22:f {NoMask}
    send (16) null  m1 null DATAPORT|RC|WRITE_SIMD16 {NoMask}     // write back to scratch
```

15.2.6 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.

Also note that IP is always an instruction-sized aligned address (16 bytes), thus the 4 LSB's are not maintained in the IP architectural register and should not be relied upon by software.

IP, when used as a source operand, reflects the memory address of the instruction in which it is used. The following are examples illustrating the use of IP:

```
      add (1) IP  4*16     // jumps to HERE_1
      add (1) IP  0x35     // jumps to HERE_1 (4 lsb's
      don't-care)
   <instruction>
   <instruction>
HERE_1:  <instruction>
HERE_2:  <instruction>
   <instruction>
   add (1) IP -2*16     // jumps to HERE_2
   ...
   add (1) IP 0      // infinite loop
   add (1) IP 0xF    // infinite loop
   ...
```

Note for Assembler: The if/iff/else/while/break instructions identify relative addresses as the targets of an implicit jump associated with the instruction. These are optional in the assembly syntax as the jitter can determine the location of the matching instruction (e.g. matching endif instruction for a given if instruction).

15.2.7 Execution Masking

15.2.7.1 Branching

Example 15-3. If / Else / EndIf

```
//-------------------------------------------------------------------
// Example if/else/endif scenario
//     "if (r5==r4) ...else ... end-if"
//-------------------------------------------------------------------
         ...
         cmp.e.f0 (8)  null r5 r4 // does r5 == r4?
         (f0) if (8)   HERE_1      // "if" part - save then update IMASK;
                                //    or goto the 'else' if all false
         ...
         ...
HERE_1:                              // now do the 'else' part
         else (8) HERE_2             // "else" part - invert IMASK
                                //    or goto the 'endif' if all false
         ...
         ...
HERE_2:
         endif                  // "end-if" part – restore IMASK
         ....              // and continue...
```

If it is known that the code has no nested conditionals, a predicate can be used for a lower overhead, more efficient if/else/endif. (One must consider the probability of all channels taking the same branch, and the number of instructions under the if/else blocks as to which conditional method, predicate or mask, is most efficient).

15.2.7.2 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 15-4. 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...
```

15.2.7.3 Cascade Branching

As there is no 'elseif' instruction, a C-like cascade branching such as if / elseif / else / endif, can be realized using the basic building blocks of if / else / endif as shown in the following example. Notice that two 'endif's' are required in order to pop the IStack correctly.

Example 15-5 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
        ....
```

15.2.7.4 Compound Branches

Compound branches are supported through the ability logically combine flag registers for each intermediate result.

Example 15-6 Compound Branch

```
//-----------------------------------------------------------------
// Example:  "if (r0 > r1) OR (r2 <= r3)"
//-----------------------------------------------------------------
        ...
        cmp.g.f0 (8)  null r0:d  r1:d       // r0 > r1?
        cmp.le.f1 (8) null r2:d  r3:d       // r2 <= r3?
        or (1) f0:w f0:w  f1:w          // combine f0 and f1
    (f0) if (8) HERE_1                  // Can now do normal if/else
        ...
        ...
HERE_1:         endif
        ...
```

Example 15-7. Compound Branch Using 'Any' or 'All'

```
//-----------------------------------------------------------------
// Example:  assuming we're doing a channel-serial vector in r0-r3
//    We want to know if all components of the vector are > 0x80
//-----------------------------------------------------------------
        ...
        cmp.g.f0 (16)   null r0 0x80         // r0 > 0x80?
        cmp.g.f1 (16)   null r1 0x80         // r1 > 0x80?
        cmp.g.f2 (16)   null r2 0x80         // r0 > 0x80?
        cmp.g.f3 (16)   null r3 0x80         // r1 > 0x80?
    (f0.all4v) if (16) HERE_1
        ...
        ...            // code executed only for those channels
        ...            // where per-channel r0,r1,r2,r3 all > 0x80
        ...
HERE_1:        endif
        ...                    //  and continue...
```

15.2.7.5 Looping

Due to Gen4'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

Example 15-8 illustrates the most basic loop. Two operations must be accomplished before loop entry. (1) Prior to loop entry, there is some subset of enabled channels as dictated by the code sequence prior. In general, the active status of each channel is indicated in the virtual EMask any point in time. These active channels will become the channels over which the loop is run, and LMask must be initialized with the EMask value. (2) Since a given loop may be nested within another loop, the previous LMask & CMask must be saved to the LStack for later restoration upon loop completion. The 'msave' instruction performs both the save and update in a single instruction, and thus all loop-blocks should be fronted with a "msave LStack LMask" and "msave LStack CMask" operation.

Note that the LMask and CMask share the same mask-stack. Thus, CMask must always be a 1's-subset of the LMask for proper stack operation. This is the case if CMask is updated to LMask each pass through the loop (see Example 15-8) 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 15-8. 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-thru
            ...
```

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 15-9. Loop Construct With Non-Nested ‘Break’

```
//------------------------------------------------------------
//   Example: While-true loop
//------------------------------------------------------------
#define BrkCode(i,d)    (i << 16) + d

            do                // save L/CMask & update
BEGIN_LOOP:
            mov (1) CMask LMask  {NoMask} // update CMask for this pass
            ...
            <some flag update>
        (<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 15-10 Loop Construct With 'Break' From Within Nested If's

```
//------------------------------------------------------------
//   Example: General Loop Structure w/ break inside if's
//------------------------------------------------------------
#define BrkCode(i,d)    (i << 16) + d

            do                      // save L/CMask & update
BEGIN_LOOP:
            mov (1) CMask LMask  {NoMask} // update CMask for this pass
            ...
            if ...
            if ...
            if ...
            ...
        (<p>)  break (8) BrkCode(3,HERE_1)  // we're 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-thru
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 Example 15-11. This save/restore operations are not required if the loop being entered does not have any occurrence of a continue instruction.

Example 15-11. 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-thru
         ...
```

15.2.7.6 Indexed Jump

Example 15-12. Indexed Jump

```
         //-----------------------------------------------------------------
    // Code example shows the use of jmpi to perform a case statement
    // of any number of options in 3 jumps
    //-----------------------------------------------------------------
    .default_execution_size      8
    ...
    jmpi r0<0,1,0>           // jump relative, based on r0.a.x
                      // ----- Jump Table ------
    jmp HERE_0               // redirect for case 0
    jmp HERE_1               // redirect for case 1
    jmp HERE_2               // redirect for case 2
    jmp HERE_3               // redirect for case 3
    ...
HERE_0:                  // ... case 0 ...
    ...
    jmp DONE
HERE_1:                  // ... case 1 ...
    ...
    jmp DONE
HERE_2:                  // ... case 2 ...
    ...
    jmp DONE
HERE_3:                  // ... case 3 ...
    ...
DONE:
    ...               //  and continue...
```