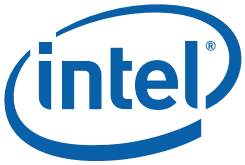


Intel® 965 Express Chipset Family and Intel® G35 Express Chipset Graphics Controller PRM

Programmer's Reference Manual (PRM)

Volume 2: 3D/Media

January 2008

Revision 1.0b

Technical queries: ilg@linux.intel.com

www.intellinuxgraphics.org

Creative Commons License

You are free:

to Share — to copy, distribute,display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

Contents

1 Introduction 13

1.1 Notations and Conventions 15

1.1.1 Reserved Bits and Software Compatibility 15

1.2 Terminology 15

2 3D Pipeline 26

2.1 Introduction 26

2.2 3D Pipeline Overview 26

2.2.1 3D Pipeline Stages 27

2.3 3D Primitives Overview 27

2.4 3D Command Overview 34

2.5 3D Pipeline State Overview 37

2.5.1 3D State Model 37

2.5.2 3DSTATE_PIPELINED_POINTERS 38

2.5.3 3DSTATE_BINDING_TABLE_POINTERS 40

2.6 Vertex Data Overview 42

2.6.1 Vertex URB Entry (VUE) Formats 42

2.6.2 Vertex Positions 44

2.6.2.1 Clip Space Position 45

2.6.2.2 NDC Space Position 45

2.6.2.3 Screen-Space Position 47

2.7 3D Pipeline Stage Overview 47

2.7.1 Generic 3D FF Unit Block Diagram 47

2.7.2 Common 3D FF Unit Functions 48

2.7.3 Pipeline Stage Input 49

2.7.4 Pipelined State Commands 50

2.7.4.1 URB_FENCE 50

2.7.4.2 3DSTATE_PIPELINED_POINTERS 50

2.7.4.3 3DSTATE_BINDING_TABLE_POINTERS 51

2.7.4.4 CONSTANT_BUFFER 51

2.7.5 Bypass Mode 52

2.7.6 URB Entry Management 52

2.7.7 Thread Initiation Management 54

2.7.7.1 Thread Input Buffering 55

2.7.7.2 Thread Resource Allocation 55

2.7.8 Thread Request Generation 57

2.7.8.1 Thread Control Information 57

2.7.8.2 Thread Payload Generation 59

2.7.9 Thread Output Handling 66

2.7.9.1 URB Entry Output (VS, GS, CLIP, SF) 67

2.7.9.2 VUE Allocation (GS, CLIP) 67

2.7.9.3 VUE Dereference (GS, CLIP) 68

2.7.9.4 Thread Termination 68

2.7.10 VUE Readback 68

2.8 Synchronization of the 3D Pipeline 69

2.8.1 End-of-Pipe Synchronization 69

2.8.2 Write Synchronization 69

2.8.3 Synchronization Actions 69
2.8.3.1 Writing a Value to Memory 69
2.8.3.2 Generating an Interrupt 70
2.8.3.3 Invalidating of Caches 70
2.8.4 PIPE_CONTROL Command 70

3 Vertex Fetch (VF) Stage 75
3.1 Vertex Fetch (VF) Stage Overview 75
3.1.1 Input Assembly 75
3.1.1.1 Vertex Assembly 75
3.1.2 Vertex Cache 76
3.2 VF Stage Input 76
3.3 Index Buffer (IB) 78
3.3.1 3DSTATE_INDEX_BUFFER 78
3.3.2 Index Buffer Access 80
3.4 Vertex Buffers (VBs) 82
3.4.1 3DSTATE_VERTEX_BUFFERS 82
3.4.2 VERTEX_BUFFER_STATE Structure 84
3.4.3 VERTEXDATA Buffers – SEQUENTIAL Access 86
3.4.4 VERTEXDATA Buffers – RANDOM Access 86
3.5 Input Vertex Definition 87
3.5.1 3DSTATE_VERTEX_ELEMENTS 88
3.5.2 VERTEX_ELEMENT_STATE Structure 89
3.5.3 Vertex Element Data Path 91
3.6 3D Primitive Processing 92
3.6.1 3DPRIMITIVE Command 92
3.6.2 Functional Overview 95
3.6.3 VertexLoop 95
3.6.4 VertexIndexGeneration 95
3.6.5 VertexCacheLookup 96
3.6.6 VertexElementLoop 97
3.6.7 SourceElementFetch 97
3.6.8 FormatConversion 97
3.6.9 DestinationFormatSelection 100
3.6.10 URBWrite 100
3.6.11 OutputBufferedVertex 101
3.7 Dangling Vertex Removal 101

4 Vertex Shader (VS) Stage 103
4.1 VS Stage Overview 103
4.1.1 Vertex Caching 103
4.2 VS Stage Input 105
4.2.1 State 105
4.2.1.1 URB_FENCE 105
4.2.1.2 VS_STATE 105
4.2.2 Input Vertices 110
4.3 VS Thread Request Generation 110
4.3.1 Thread Payload 112
4.4 VS Thread Execution 114
4.4.1 Vertex Output 114
4.4.2 Thread Termination 114
4.5 Primitive Output 114

5 Geometry Shader (GS) Stage 116
5.1 GS Stage Overview 116
5.2 GS Stage Input 116
5.2.1 State 117
5.2.1.1 GS_STATE 117
5.3 Object Staging 123
5.4 GS Thread Request Generation 123
5.4.1 Object Vertex Ordering 123
5.4.2 GS Thread Payload 123
5.5 GS Thread Execution 126
5.5.1 Vertex Output 126
5.5.2 Thread Termination 128
5.6 Vertex Header Readback 128
5.7 Primitive Output 128

6 Clip Stage 129
6.1 CLIP Stage Overview 129
6.1.1 Clip Stage – General-Purpose Processing 129
6.1.2 Clip Stage – 3D Clipping 129
6.2 Concepts 130
6.2.1 The Clip Volume 130
6.2.1.1 View Volume 130
6.2.2 User-Specified Clipping 133
6.2.2.1 User Clip Planes 133
6.2.3 Negative-W Clipping Errata 133
6.2.3.1 W Clipping Errata (DevBW, DevCL-A) 135
6.2.3.2 W Clipping Errata (DevCL-B) 136
6.2.4 Tristrip Clipping Errata [Pre-DevBW-E1], [DevCL] 138
6.2.5 Guard Band 139
6.2.5.1 NDC Guardband Parameters 142
6.2.5.2 Screen Space Guardband Parameters 142
6.2.6 Vertex-Based Clip Testing & Considerations 143
6.2.6.1 Triangle Objects 143
6.2.6.2 Non-Wide Line Objects 143
6.2.6.3 Wide Line Objects 144
6.2.6.4 Wide Points 144
6.2.6.5 RECTLIST 145
6.2.7 3D Clipping 145
6.3 CLIP Stage Input 146
6.3.1 State 146
6.3.1.1 CLIP_STATE 146
6.3.1.2 CLIP_VIEWPORT 153
6.4 VertexClipTest Function 155
6.5 Object Staging 159
6.5.1 Partial Object Removal 159
6.5.2 ClipDetermination Function 160
6.5.3 ClipMode 163
6.5.3.1 NORMAL ClipMode 164
6.5.3.2 CLIP_ALL ClipMode 164
6.5.3.3 CLIP_NON_REJECT ClipMode 164
6.5.3.4 REJECT_ALL ClipMode 164
6.5.3.5 ACCEPT_ALL ClipMode 164
6.6 Object Pass-Through 165

6.7 CLIP Thread Request Generation 166
6.7.1 Object Vertex Ordering 166
6.7.2 CLIP Thread Payload 168
6.8 CLIP Thread Execution 170
6.8.1 Vertex Output 171
6.8.2 Thread Termination 171
6.9 Thread-Generated Vertex Readback 172
6.10 Primitive Output 172
6.11 Other Functionality 173
6.11.1 Statistics Gathering 173
6.11.1.1 CL_INVOCATION_COUNT 173
6.11.1.2 GS_PRIMITIVES_COUNT 173

7 Strips and Fans (SF) Stage 175

7.1 Overview 175
7.1.1 Inputs from CLIP 175
7.1.2 Attribute Setup/Interpolation Process 176
7.1.3 Outputs to WM 177
7.2 Primitive Assembly 177
7.2.1 Point List Decomposition 181
7.2.2 Line List Decomposition 182
7.2.3 Line Strip Decomposition 182
7.2.4 Triangle List Decomposition 184
7.2.5 Triangle Strip Decomposition 185
7.2.6 Triangle Fan Decomposition 186
7.2.7 Polygon Decomposition 186
7.2.8 Rectangle List Decomposition 187
7.3 Object Setup 187
7.3.1 Invalid Position Culling (Pre/Post-Transform) 187
7.3.2 Viewport Transformation 188
7.3.3 Destination Origin Bias 188
7.3.4 Point Rasterization Rule Adjustment 189
7.3.5 Drawing Rectangle Offset Application 191
7.3.5.1 3DSTATE_DRAWING_RECTANGLE 193
7.3.6 Point Width Application 195
7.3.7 Rectangle Completion 196
7.3.8 Vertex X,Y Clamping and Quantization 196
7.3.9 Degenerate Object Culling 197
7.3.10 Degenerate Triangle Culling 197
7.3.11 Triangle Orientation (Face) Culling 197
7.3.12 Scissor Rectangle Clipping 198
7.3.13 Line Rasterization 199
7.3.13.1 Zero-Width (Cosmetic) Line Rasterization 199
7.3.13.2 Diamond Exit Sampling Rules – Legacy Mode 200
7.3.13.3 Diamond Exit Sampling Rules – New Mode 203
7.3.13.4 Non-Antialiased Wide Line Rasterization 204
7.3.13.5 Anti-aliased Line Rasterization 205
7.4 SF Pipeline State Summary 207
7.4.1 SF_STATE 207
7.4.2 SF_VIEWPORT 215
7.5 The SF Thread -- Interpolation Coefficient Calculation 216
7.5.1 SF Setup Parameters Passed to SF Thread 216
7.5.1.1 TRIANGLE Parameters 216

7.5.1.2 RECTANGLE Parameters 216
7.5.1.3 POINT Parameters 216
7.5.1.4 LINE Parameters 217
7.5.2 SF (Setup) Thread Payload 217
7.5.3 SF Thread Execution 220
7.5.4 SF Thread Output 220
7.6 Other SF Functions 222
7.6.1 Statistics Gathering 222

8 Windower (WM) Stage 224
8.1 Overview 224
8.1.1 Inputs from SF to WM 225
8.2 Windower Pipelined State 225
8.2.1 WM_STATE 225
8.3 Rasterization 232
8.3.1 Drawing Rectangle Clipping 232
8.3.2 Line Rasterization 233
8.3.2.1 Coverage Values for Anti-Aliased Lines 233
8.3.2.2 Line Stipple 233
8.3.2.3 3DSTATE_LINE_STIPPLE 234
8.3.3 Polygon (Triangle and Rectangle) Rasterization 235
8.3.3.1 Polygon Stipple 236
8.3.3.2 3DSTATE_POLY_STIPPLE_OFFSET 237
8.3.3.3 3DSTATE_POLY_STIPPLE_PATTERN 238
8.3.3.4 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP 239
8.4 Early Depth/Stencil Processing 240
8.4.1 Depth Coefficient Read-Back 240
8.4.2 Depth Offset 240
8.4.3 Early Depth Test / Stencil Test/Write 241
8.4.3.1 Software-Provided PS Kernel Info 241
8.4.3.2 Early Depth Test Cases 242
8.4.4 Depth/Stencil Buffer State 249
8.4.4.1 3DSTATE_DEPTH_BUFFER 249
8.5 Pixel Shader Thread Generation 257
8.5.1 Pixel Grouping (Dispatch Size) Control 258
8.5.2 PS Thread Payload for Normal Dispatch 260
8.6 Other WM Functions 269
8.6.1 Statistics Gathering 269

9 Color Calculator (Output Merger) 270
9.1.1 Alpha Test 271
9.1.2 Depth Buffer Coordinate Offset Disable 271
9.1.3 Stencil Test 273
9.1.4 Depth Test 273
9.1.5 Pre-Blend Color Clamping 274
9.1.6 Color Buffer Blending 275
9.1.6.1 3DSTATE_CONSTANT COLOR 278
9.1.7 Post-Blend Color Clamping 279
9.1.8 Color Quantization 279
9.1.9 Dithering 279
9.1.10 Buffer Update 280
9.1.10.1 Stencil Buffer Updates 280
9.1.10.2 Depth Buffer Updates 281
9.1.10.3 Color Gamma Correction 282

9.1.10.4 Color Buffer Updates 282
9.2 Pixel Pipeline State Summary 283
9.2.1 COLOR_CALC_STATE 283
9.2.2 CC_VIEWPORT 293
9.3 Other Pixel Pipeline Functions 293
9.3.1 Statistics Gathering 293

10 Media and General Purpose Pipeline 295
10.1 Introduction 295
10.1.1 Terminologies 296
10.2 Media Pipeline Overview 298
10.3 Programming Media Pipeline 299
10.3.1 Command Sequence 299
10.3.2 Interrupt Latency 304
10.4 Video Front End Unit 305
10.4.1 Interfaces 306
10.4.1.1 Interface to Command Streamer 306
10.4.1.2 Interface to Thread Spawner 306
10.4.1.3 Interface to State Variable Manager 306
10.4.1.4 Interface to Global URB Manager 306
10.4.1.5 Interface to URB 307
10.4.2 Mode of Operations 307
10.4.2.1 Generic Mode 307
10.4.2.2 IS Mode 308
10.4.2.3 VLD Mode 308
10.4.3 Debug Counter 318
10.5 Thread Spawner Unit 320
10.5.1 Basic Functions 320
10.5.1.1 Root Threads Lifecycle 320
10.5.1.2 URB Handles 321
10.5.1.3 Root to Child Responsibilities 321
10.5.1.4 Multiple Simultaneous Roots 322
10.5.1.5 Synchronized Root Threads 323
10.5.1.6 Deadlock Prevention 323
10.5.1.7 Child Thread Lifecycle 324
10.5.1.8 Arbitration between Root and Child Threads 325
10.5.2 Interfaces 325
10.5.2.1 Interface to VFE 325
10.5.2.2 Interface to Thread Dispatcher 325
10.6 Media State 326
10.6.1 Media State Model 326
10.6.2 VFE_STATE 327
10.6.3 VLD_STATE 329
10.6.4 INTERFACE_DESCRIPTOR 331
10.7 Media State and Primitive Commands 333
10.7.1 MEDIA_STATE_POINTERS Command 333
10.7.2 MEDIA_OBJECT Command 335
10.7.2.1 Inline and Indirect Data Format in Generic Mode 337
10.7.2.2 Inline and Indirect Data Format in IS Mode 337
10.7.2.3 Inline and Indirect Data Format in VLD Mode 343
10.8 Media Messages 344
10.8.1 Thread Payload Messages 344
10.8.1.1 Generic Mode Root Thread 345

10.8.1.2 IS-Mode Root Thread 346
10.8.1.3 VLD-Mode Root Thread 351
10.8.1.4 Child Thread 356
10.8.2 Thread Spawn Message 357
10.8.2.1 Message Descriptor 358
10.8.2.2 Message Payload 359
10.9 Media Applications with Specific Hardware Support 360
10.9.1 Full MPEG-2 Decode 360
10.9.1.1 Theory of Operation 360
10.9.1.2 Performance 364
10.10 Media Kernel Design Guide 364
10.10.1 MPEG-2 HWMC 364
10.10.2 Deinterlace Filter 366
10.10.3 Video Encode 366

Figures

Figure 6-1. SW Workaround Summary 135
Figure 6-2. Normal Guardband Operation 140
Figure 6-3. Very Large Viewport Case 141
Figure 7-1. 3DPRIM_POINTLIST Primitive 181
Figure 7-2. 3DPRIM_LINELIST Primitive 182
Figure 7-3. 3DPRIM_LINESTRIP_xxx Primitive 183
Figure 7-4. 3DPRIM_TRILIST Primitive 184
Figure 7-5. 3DPRIM_TRISTRIP[_REVERSE] Primitive 185
Figure 7-6. 3DPRIM_TRIFAN Primitive 186
Figure 7-7. 3DPRIM_RECTLIST Primitive 187
Figure 7-8. Destination Origin Bias 189
Figure 7-9. RASTRULE_UPPER_LEFT 190
Figure 7-10. RASTRULE_UPPER_RIGHT 191
Figure 7-11. Onscreen Draw Rectangle 192
Figure 7-12. Partially-offscreen Draw Rectangle 192
Figure 7-13. Point Width Application 195
Figure 7-14. Rectangle Completion 196
Figure 7-15. Triangle Winding Order 198
Figure 7-16. Non-Antialiased Line Rasterization 205
Figure 7-17. Anti-aliased Line Rasterization 206
Figure 8-1. Pixels with a SubSpan 232
Figure 8-2. TRIANGLE and RECTANGLE Edge Types 236
Figure 9-1. Drawing Rectangle Offset 272
Figure 9-2. Dithering Process (5-Bit Example) 280
Figure 10-1. Top level block diagram of the Media Pipeline 299
Figure 10-2. VFE Functional Blocks and Modes of Operations 305
Figure 10-3. Prediction for a P field picture that is a first field, which is (a) a top field, or (b) a bottom field. 311
Figure 10-4. Prediction for a P field picture that is a second field, which is (a) a top field, or (b) a bottom field 311
Figure 10-5. Thread Spawner block diagram 320
Figure 10-6. Examples of thread relationship 322
Figure 10-7. An example of thread relationship with root sibling dependency 322
Figure 10-8. Media State Model 326
Figure 10-9. Structure of the IDCT Compressed Data Buffer 342
Figure 10-10. Indirect data buffer for a slice 344
Figure 10-11. Thread payload message formats for root and child threads 344
Figure 10-12. MPEG-2 decode flow chart 361
Figure 10-13. MPEG-2 compressed bitstream syntax 362
Figure 10-14. Functional mapping of MPEG-2 decode hardware acceleration with off-host VLD 363
Figure 10-15. Functional mapping of MPEG-2 decode hardware acceleration with HWMC 365

Tables

Table 1-1. Supported Chipsets 13
Table 2-1. 3D Primitive Topology Types 28
Table 2-2. VUE Vertex HeaderVUE Vertex Header 43
Table 2-3. State Variables Included in Thread Control Information 58
Table 2-4. Payload Sizes 59
Table 2-5. Fixed Payload Header Fields (non-FF-specific) 62
Table 2-6. State Variables Controlling Payload URB Data 65
Table 2-7. Caches Invalidated/Flushed by PIPE_CONTROL Bit Settings 71
Table 3-1. 3D Primitive Topology Type Encoding 94
Table 3-2. Source Element Formats supported in VF Unit 97
Table 4-1. VS Thread Payload 112
Table 5-1. GS Thread Payload 123
Table 6-1. CLIP Thread Payload 168
Table 7-1. SF's Vertex Pipeline Inputs 175
Table 7-2. SF-Supported Primitive Types & Vertex Count Restrictions 178
Table 7-3. 3D Object Types 178
Table 7-4. Primitive Decomposition Outputs 179
Table 7-5. VPIndex/RTAIndex Selection 180
Table 7-6. Degenerate Objects 197
Table 7-7. Cull Mode 198
Table 8-1. Variable Pixel Dispatch 259
Table 9-1. Color Buffer Blend Color Factors 277
Table 9-2. Color Buffer Blend Functions 278
Table 9-3. Stencil Buffer Operations 281
Table 10-1. Summary of Motion Types 312
Table 10-2. Motion Comp Operation for Pictures with Frame Motion Type 313
Table 10-3. Motion Comp Operation with Field Motion Type 314
Table 10-4. Converting Frame-Dual Prime Motion to 4MV 315
Table 10-5. Converting Field-Dual Prime Motion to 2MV 315
Table 10-6. Macroblock indices for frame picture destination 317
Table 10-7. Macroblock indices for field picture destination 317
Table 10-8. TS Resource Available in Device Hardware 324
Table 10-9. Inline data in IS mode 337
Table 10-10. Subblock coding (bits [7:6] are reserved). 341
Table 10-11. Structure of a DCT coefficient unit 342
Table 10-12. R0 header of a generic mode root thread 345
Table 10-13. Format of a block of DCT coefficients in GRF registers 355
Table 10-14. Use of GEN4 shared resources for post-VLD kernels 363
Table 10-15. Use of GEN4 shared resources for HWMC kernels 365

Revision History

Document Number	Revision Number	Description	Revision Date
2	1.0b	Initial release.	January 2008

§§

1 Introduction

This Programmer's Reference Manual (PRM) describes the architectural behavior and programming environment of the Intel® 965 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.

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.
In-loop Deblocking Filter	ILDB	The deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe.
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.
Media Pipeline	—	Fixed function stages dedicated to media and "generic" processing, sometimes referred to as the generic pipeline.

Term	Abbr.	Definition
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.
Primitive Topology	—	A composite primitive such as a triangle strip, or line list. Also includes the objects triangle, line and point as degenerate cases.

Term	Abbr.	Definition
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 a 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.
Shared Function ID	SFID	Unique identifier used by kernels and shaders to target shared functions and to identify their returned messages.

Term	Abbr.	Definition
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.
Synchronized Root Thread	—	A root thread that is dispatched by TS upon a 'dispatch root thread' message.

Term	Abbr.	Definition
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

Term	Abbr.	Definition
URB Entry	UE	URB Entry: A logical entity stored in the URB (such as a vertex), referenced via a URB Handle.
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.
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.

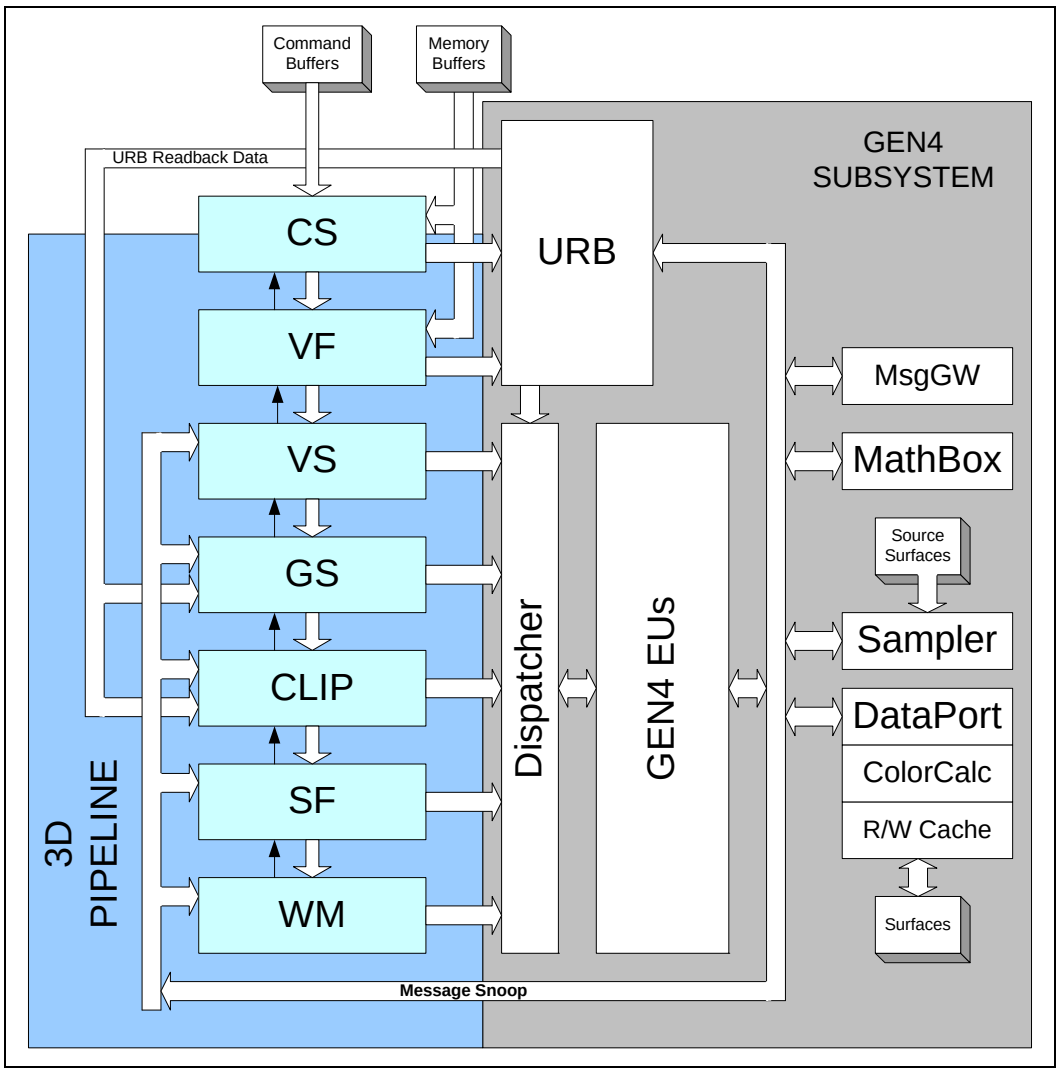
Term	Abbr.	Definition
Vertical Stride	VertStride	The distance in element-sized units between 2 vertically-adjacent elements of a Gen4 region-based GRF access.
Video Front End	VFE	The first fixed function in the GEN4 generic pipeline; performs fixed-function media operations.
Viewport	VP	Post-clipped 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 3D Pipeline

2.1 Introduction

This section covers the programming details for the 3D fixed functions.

2.2 3D Pipeline Overview



2.2.1 3D Pipeline Stages

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

Pipeline Stage	Functions Performed
• Command Stream • (CS)	• The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. • Note that the CS stage is shared between the 3D and Media pipelines.
• Vertex Fetch • (VF)	• The Vertex Fetch stage, in response to 3D Primitive Processing commands, is responsible for reading vertex data from memory, reformatting it, and writing the results into Vertex URB Entries. It then outputs primitives by passing references to the VUEs down the pipeline.
• Vertex Shader (VS)	• The Vertex Shader stage is responsible for processing (shading) incoming vertices by passing them to VS threads.
• Geometry Shader (GS)	• The Geometry Shader stage is responsible for processing incoming objects by passing each object's vertices to a GS thread.
• Clipper • (CLIP)	• The Clipper stage performs clip test on incoming objects and, if required, clips objects via CLIP threads.
• Strip/Fan • (SF)	• The Strip/Fan stage performs object setup via use of spawned SF threads (aka Setup threads).
• Windower/Masker • (WM)	• The Windower/Masker performs object rasterization and spawns WM thread (aka PS thread) to process (shade) the object pixels.

2.3 3D Primitives Overview

The 3DPRIMITIVE command (defined in the *VF Stage* chapter) is used to submit 3D primitives to be processed by the 3D pipeline. Typically the processing results in the rendering of pixel data into the render targets, but this is not required.

Note: ***Terminology Note*****:** There is considerable confusion surrounding the term 'primitive', e.g., is a triangle strip a 'primitive', or is a triangle within a triangle strip a 'primitive'? The D3D10 Spec introduces the term 'topology' to describe the higher-level construct (e.g., a triangle strip), and uses the term 'primitive' when discussing a triangle within a triangle strip. In this spec, we will try to avoid ambiguity by using the term 'object' to represent the basic shapes (point, line, triangle), and 'topology' to represent input geometry (strips, lists, etc.). Unfortunately, terms like '3DPRIMITIVE' must remain for legacy reasons.

The following table describes the basic primitive topology types supported in the 3D pipeline.

Notes:

- There are several variants of the basic topologies. These have been introduced to allow slight variations in behavior without requiring a state change.
- Number of vertices:
  - **Dangling Vertices**: Topologies have an "expected" number of vertices in order to form complete objects within the topologies (e.g., LINELIST is expected to have an even number of vertices). The actual number of vertices specified in the 3DPRIMITIVE command, and as output from the GS unit, is allowed to deviate from this expected number --- in which case any "dangling" vertices are discarded. The removal of dangling vertices is initially performed in the VF unit. In order to filter out dangling vertices emitted by GS threads, the CLIP unit also performs dangling-vertex removal at its input. However, the CLIP unit is required to output the expected number (based on the assumption that the clipping kernel is thoroughly validated). If a CLIP thread violates this restriction, behavior is UNDEFINED.

Table 2-1. 3D Primitive Topology Types

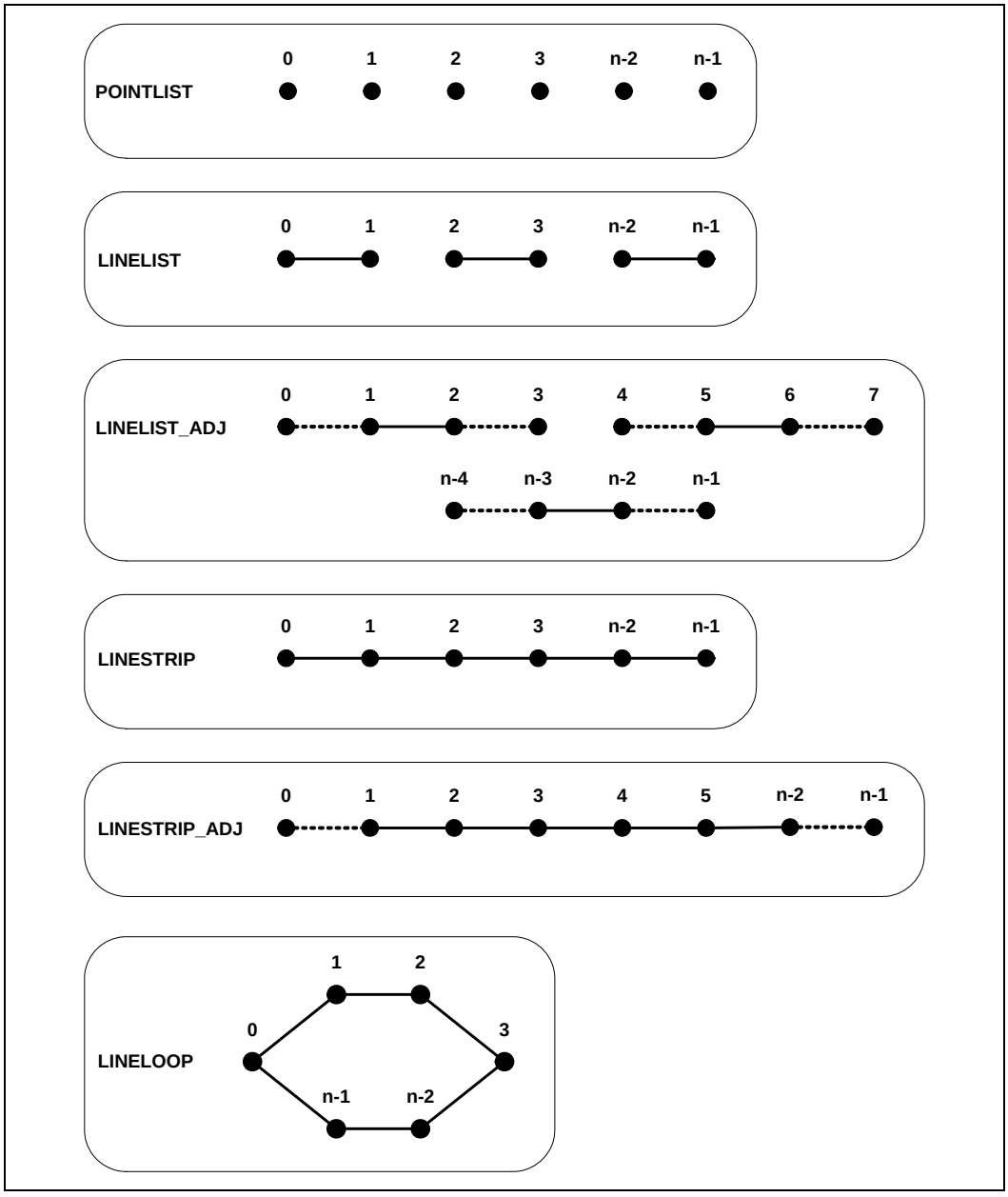
3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	A list of independent line objects (2 vertices per line). **Programming Restrictions:** • Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELOOP	Similar to a 3DPRIM_LINESTRIP, though the last vertex is connected back to the initial vertex via a line object. **Programming Restrictions:** • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored. (The 2-vertex case is required by OGL). • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
LINESTRIP	A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define a connected line object. **Programming Restrictions:** • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored.
LINESTRIP_BF	Similar to LINESTRIP, except treated as "backfacing' during rasterization (stencil test). This can be used to support "line" polygon fill mode when two-sided stencil is enabled.

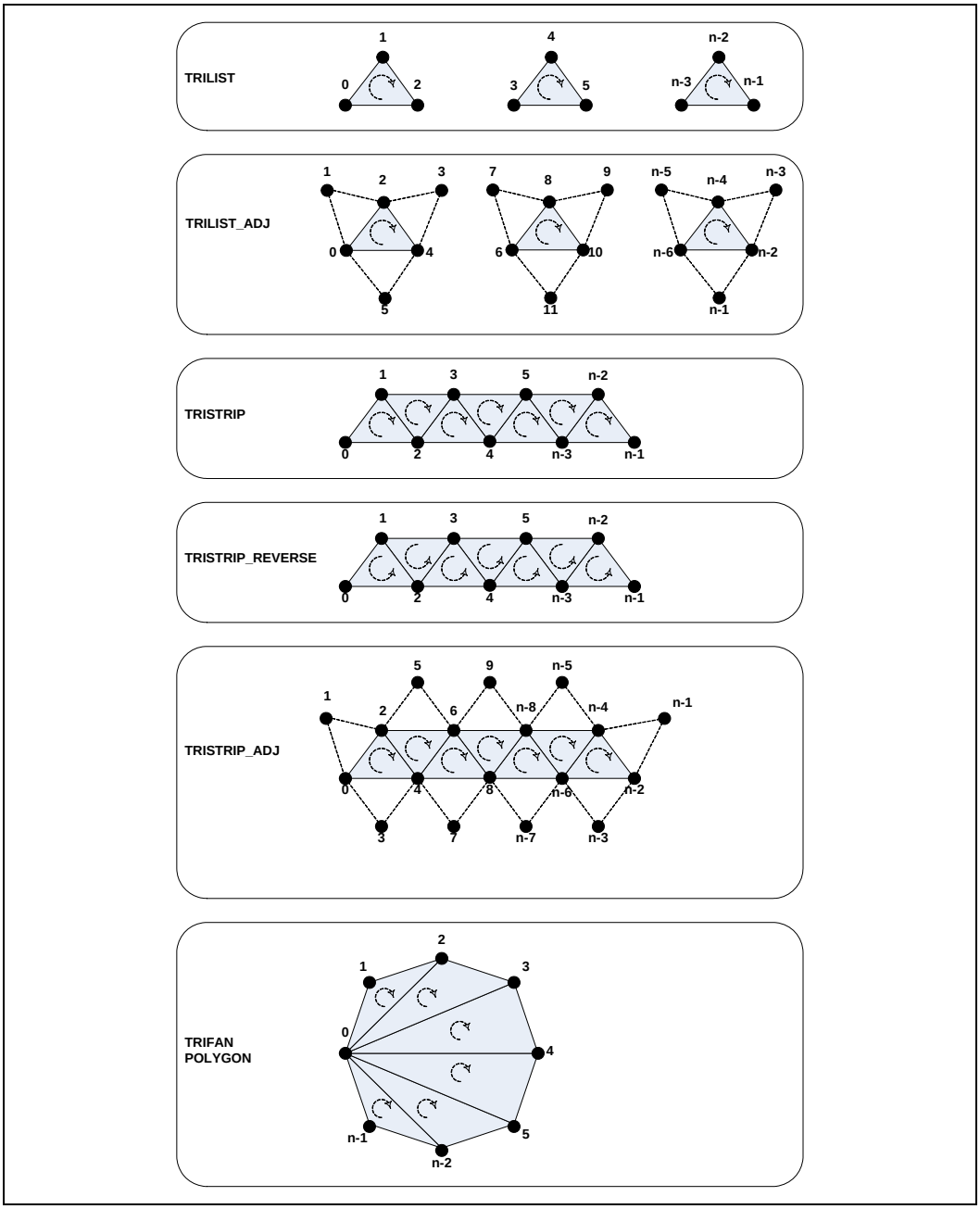
3D Primitive Topology Type (ordered alphabetically)	Description
LINESTRIP_CONT	Similar to LINESTRIP, except LineStipple (if enabled) is continued (vs. reset) at the start of the primitive topology. This can be used to support line stipple when the API-provided primitive is split across multiple tolopologies.
LINESTRIP_CONT_BF	Combination of LINESTRIP_BF and LINESTRIP_CONT variations.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	Similar to POINTLIST, except treated as “backfacing’ during rasterization (stencil test). This can be used to support “point” polygon fill mode when two-sided stencil is enabled.
POLYGON	Similar to TRIFAN, though the first vertex always provides the “flat-shaded” values (vs. this being programmable through state). **Programming Restrictions:** • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
QUADLIST	A list of independent quad objects (4 vertices per quad). **Programming Restrictions:** • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
QUADSTRIP	A list of vertices connected such that, after the first two vertices, each additional pair of vertices are associated with the previous two vertices to define a connected quad object. **Programming Restrictions:** • Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored. • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).

3D Primitive Topology Type (ordered alphabetically)	Description
RECTLIST	A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle. V0=LowerRight, V1=LowerLeft, V2=UpperLeft. Implied V3 = V0-V1+V2. **Programming Restrictions:** • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored. • The RECTLIST primitive is supported specifically for 2D operations (e.g., BLTs and "stretch" BLTs) and not as a general 3D primitive. Due to this, a number of restrictions apply to the use of RECTLIST: — Must utilize "screen space" coordinates (VPOS_SCREENSPACE) when the primitive reaches the CLIP stage. The W component of position must be 1.0 for all vertices. The 3 vertices of each object should specify a screen-aligned rectangle (after the implied vertex is computed). — Clipping: Must not require clipping or rely on the CLIP unit's ClipTest logic to determine if clipping is required. Either the CLIP unit should be DISABLED, or the CLIP unit's **Clip Mode** should be set to a value other than CLIPMODE_NORMAL. — Viewport Mapping must be DISABLED (as is typical with the use of screen-space coordinates).
TRIFAN	Triangle objects arranged in a fan (or polygon). The initial vertex is maintained as a common vertex. After the second vertex, each additional vertex is associated with the previous vertex and the common vertex to define a connected triangle object . **Programming Restrictions:** • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRIFAN_NOSTIPPLE	Similar to TRIFAN, but poylgon stipple is not applied (even if enabled). This can be used to support "point" polygon fill mode, under the combination of the following conditions: (a) when the frontfacing and backfacing polygon fill modes are different (so the final fill mode is not known to the driver), (b) one of the fill modes is "point" and the other is "solid", (c) point mode is being emulated by converting the point into a trifan, (d) polygon stipple is enabled. In this case, polygon stipple should not be applied to the points-emulated-as-trifans.
TRILIST	A list of independent triangle objects (3 vertices per triangle). **Programming Restrictions:** • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.

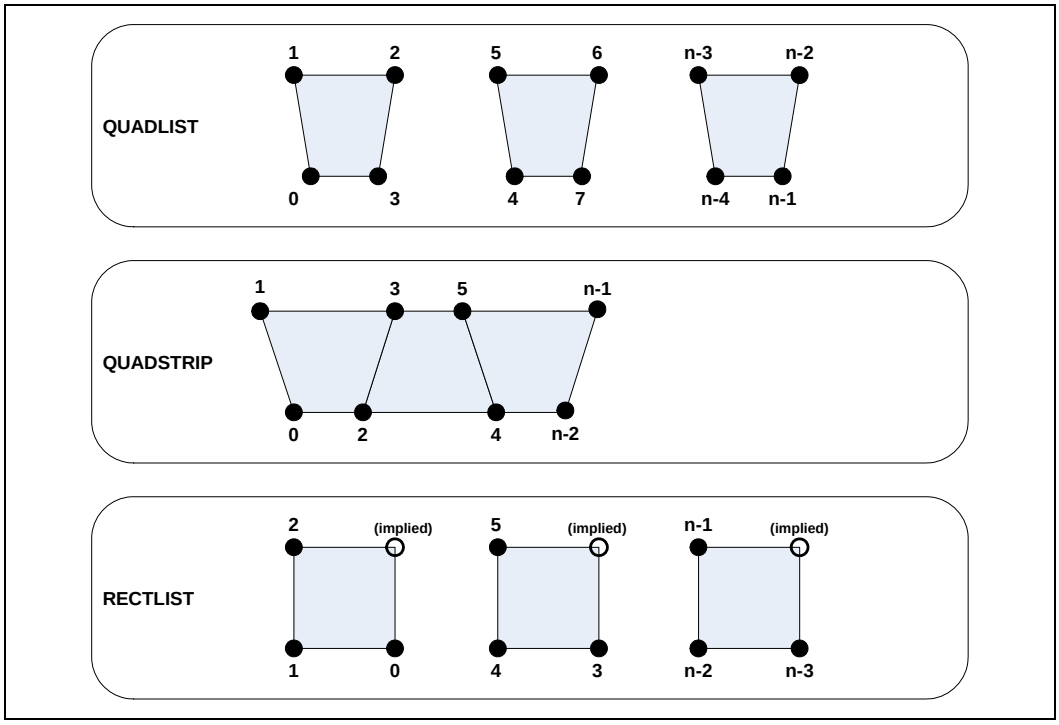
3D Primitive Topology Type (ordered alphabetically)	Description
TRISTRIP	A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. **Programming Restrictions:** • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRISTRIP_REVERSE	Similar to TRISTRIP, though the sense of orientation (winding order) is reversed – this allows SW to break long tristrips into smaller pieces and still maintain correct face orientations.

The following diagrams illustrate the basic 3D primitive topologies. (Variants are not shown if they have the same definition with respect to the information provided in the diagrams).





A note on the arrows you see below: These arrows are intended to show the vertex ordering of triangles that are to be considered having “clockwise” winding order in screen space. Effectively, the arrows show the order in which vertices are used in the cross-product (area, determinant) computation. Note that for TRISTRIP, this requires that either the order of odd-numbered triangles be reversed in the cross-product or the sign of the result of the normally-ordered cross-product be flipped (these are identical operations).



2.4 3D Command Overview

The following table lists and summarizes the commands supported by the 3D Pipeline.

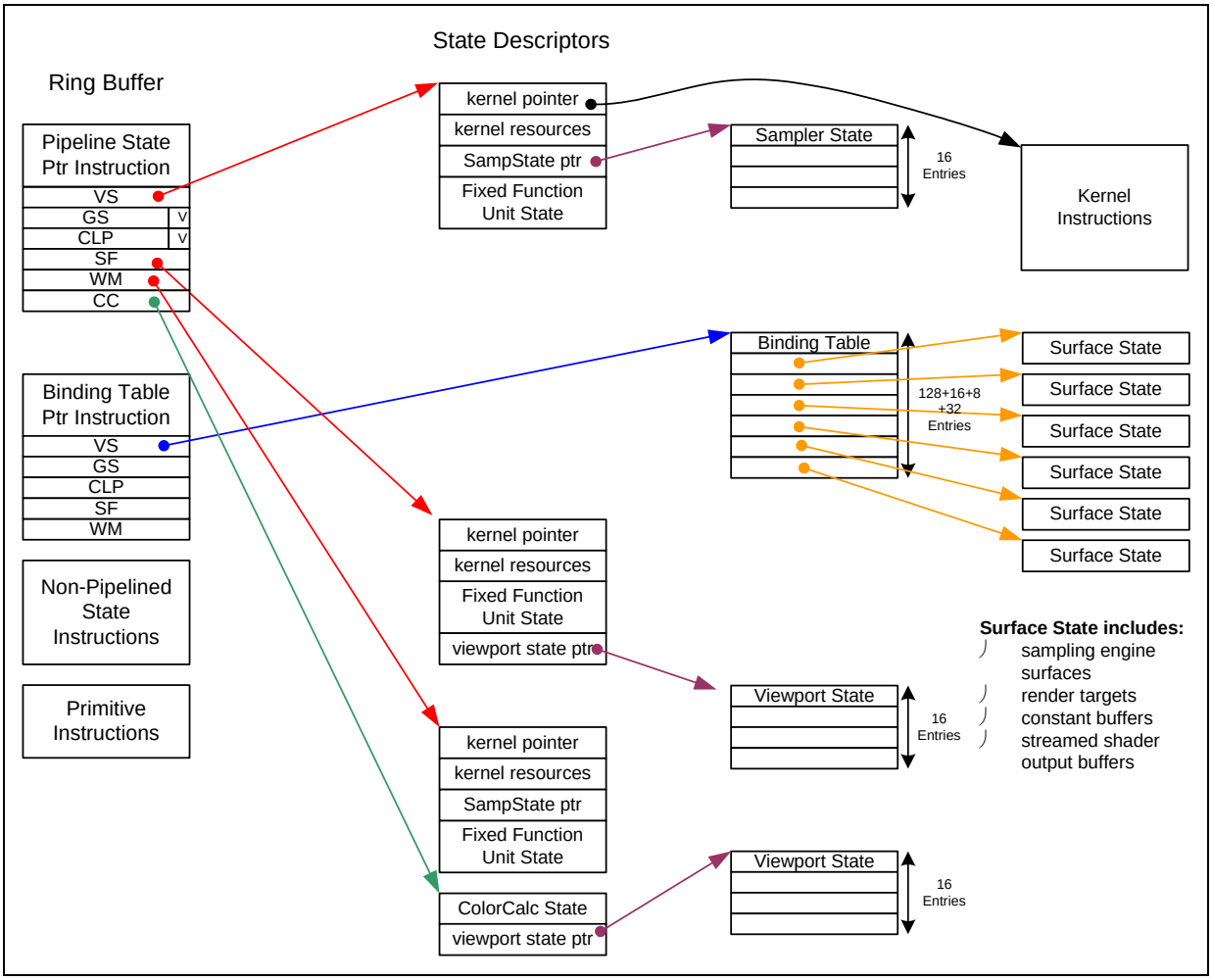
Command	Description
Processing Commands	
3DPRIMITIVE	This primitive command is used to inject primitives into the 3D pipeline, where they will be processed according to the current context state settings. Most typically this processing will result in rendering to destination surfaces, though this is not required. This command is defined in the *VF Stage* chapter (as it is executed there), though the processing of this command includes the entire 3D pipeline.

Command	Description
Control Operation Commands	
PIPE_CONTROL	This control operation command allows software to synchronize 3D pipeline operations as seen by the CPU. For example, this command can be used to inform the CPU (via a CPU-snoopable memory write or CPU interrupt) when previously-issued commands have reached a certain point, such as read operations complete or results coherent in memory. This command is described later in this chapter.
STATE_PREFETCH	This control operation command allows software to initiate the prefetch of memory data into the pipeline's Instruction and State Cache. This command is provided solely for performance optimization. See Graphics Processing Engine chapter.
Pipelined State Commands	
URB_FENCE	This pipelined state command is used to allocate regions of the URB between the FF units of the 3D and Media Pipelines.
3DSTATE_PIPELINED_POINTERS	This pipelined state command is used to provide the 3D FF units with offsets to Pipeline State Blocks stored in memory. These state blocks are read by the FF units and supply the bulk of the state variable settings which control the operation of the units. This command is described later in this chapter.
3DSTATE_BINDING_TABLE_POINTERS	This pipelined state command is used to provide the 3D FF units with offsets to Binding Tables stored in memory. The Binding Tables are not directly accessed by the FF units, instead the pointers are passed in thread payloads and eventually routed to the GEN4 shared functions where they are used to access memory surfaces. This command is described later in this chapter.
CS_URB_STATE	This pipelined state command is used to define the number and size of URB entries owned by the CS stage (for use as Constant URB Entries). See Graphics Processing Engine chapter.
CONSTANT_BUFFER	This pipelined state command is used to define a region of memory that contains "constant" parameters to be passed to threads. The constants are read from memory, stored in the URB, and supplied in thread payloads. See Graphics Processing Engine chapter.

Command	Description
3DSTATE_INDEX_BUFFER	This pipelined state command is used to specify Index Buffer parameters used in the VF unit's InputAssembly function. An Index Buffer can be used to provide vertex indices when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See VF Stage chapter.
3DSTATE_VERTEX_BUFFERS	This pipelined state command is used to specify Vertex Buffer parameters used in the VF unit's InputAssembly function. Vertex Buffers provide vertex data when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See VF Stage chapter.
3DSTATE_VERTEX_ELEMENTS	This pipelined state command is used to specify Vertex Element parameters used in the VF unit's InputAssembly function. Vertex Element parameters specify how vertex data, extracted from Vertex Buffers, are format converted and stored in VUEs. This command does not travel past the VF stage. See VF Stage chapter.
3DSTATE_SAMPLER_CACHE_DISABLE	This pipelined state command is used to control Texture Cache operation. See Sampler chapter.
Non-Pipelined State Commands	
STATE_BASE_ADDRESS	This pipelined state command is used to supply base memory addresses used by various functions to access non-surface memory operands (e.g., GEN4 instructions, pipeline state, binding tables, sampler state, etc.) See Graphics Processing Engine chapter.
3DSTATE_SAMPLER_PALETTE_LOAD	This non-pipelined state command is used to load the Texture Palette state used by the Sampler shared function. See Sampler chapter.

2.5 3D Pipeline State Overview

2.5.1 3D State Model



2.5.2 3DSTATE_PIPELINED_POINTERS

The 3DSTATE_PIPELINED_POINTERS command is used to set up the pointers to the 3D fixed function state. It is also used to disable the GS and/or CLIP units and make them pass-through (input flows through to output). The other units are (by definition) "enabled", meaning they will fetch and use the associated pipelined state to control the unit's functions.

[DevBW-A,B] Errata BWT007: State data pointed at by offsets from **General State Base** must be contained within 32-bit physical address space (that is, must entirely map to memory pages under 4GB.)

3DSTATE_PIPELINED_POINTERS

Project:	All	**Length Bias:**	2

The 3DSTATE_PIPELINED_POINTERS command is used to set up the pointers to the 3D fixed function state. It is also used to disable the GS and/or CLIP units and make them pass-through (input flows through to output). The other units are (by definition) "enabled", meaning they will fetch and use the associated pipelined state to control the unit's functions.

[DevBW-A,B] Errata BWT007: State data pointed at by offsets from **General State Base** must be contained within 32-bit physical address space (that is, must entirely map to memory pages under 4GB.)

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: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 00h 3DSTATE_PIPELINED_POINTERS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 5h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:5	**Pointer to VS_STATE** Project: All Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of the VS_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

3DSTATE_PIPELINED_POINTERS		
2	31:5	**Pointer to GS_STATE** Project: All Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of the GS_STATE. This offset is relative to the **General State Base Address**.
	4:1	**Reserved** Project: All Format: MBZ
	0	**GS Enable** Project: All Format: Enable FormatDesc Specifies whether the GS function is enabled or disabled (pass-through). If this bit is set to DISABLED, the pointer to GS_STATE is ignored. **Programming Note:** When enabling the GS stage that may generate incomplete objects, the CLIP stage also needs to be ENABLED in order to filter out any incomplete objects. See *Clipper* chapter.
3	31:5	**Pointer to CLIP_STATE** Project: All Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of the CLIP_STATE. This offset is relative to the **General State Base Address**.
	4:1	**Reserved** Project: All Format: MBZ
	0	**CLIP Enable** Project: All Format: Enable FormatDesc Specifies whether the CLIP function is enabled or disabled (pass-through). If this bit is set to ENABLED, the pointer to CLIP_STATE is ignored. **Programming Note:** When enabling the GS stage that may generate incomplete objects, the CLIP stage also needs to be ENABLED in order to filter out any incomplete objects. See *Clipper* chapter.
4	31:5	**Pointer to SF_STATE** Project: All Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of the SF_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

3DSTATE_PIPELINED_POINTERS		
5	31:5	**Pointer to WM_STATE** Project: All Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of the WM_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
6	31:6	**Pointer to COLOR_CALC_STATE** Project: All Format: GeneralStateOffset[31:6] FormatDesc Specifies the 64-byte aligned offset of the COLOR_CALC_STATE. This offset is relative to the **General State Base Address**.
	5:0	**Reserved** Project: All Format: MBZ

2.5.3 3DSTATE_BINDING_TABLE_POINTERS

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

The 3DSTATE_BINDING_TABLE_POINTERS command is used to define the location of fixed functions' BINDING_TABLE_STATE. Only some of the fixed functions utilize binding tables.

[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 entirely map to memory pages under 4G.)

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: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h 3DSTATE_BINDING_TABLE_POINTERS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 4h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All

3DSTATE_BINDING_TABLE_POINTERS		
1	31:5	**Pointer to VS Binding Table** Project: All Format: SurfaceStateOffset[31:5] FormatDesc Specifies the 32-byte aligned address offset of the VS function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
2	31:5	**Pointer to GS Binding Table** Project: All Format: SurfaceStateOffset[31:5] FormatDesc Specifies the 32-byte aligned address offset of the GS function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
3	31:5	**Pointer to CLIP Binding Table** Project: All Format: SurfaceStateOffset[31:5] FormatDesc Specifies the 32-byte aligned address offset of the CLIP function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
4	31:5	**Pointer to SF Binding Table** Project: All Format: SurfaceStateOffset[31:5] FormatDesc Specifies the 32-byte aligned address offset of the SF function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
5	31:5	**Pointer to PS Binding Table** Project: All Format: SurfaceStateOffset[31:5] FormatDesc Specifies the 32-byte aligned address offset of the PS (Windower) function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

2.6 Vertex Data Overview

The 3D pipeline FF stages (past VF) receive input 3D primitives as a stream of vertex information packets. (These packets are not directly visible to software). Much of the data associated with a vertex is passed indirectly via a VUE handle. The information provided in vertex packets includes:

- The **URB Handle** of the VUE: This is used by the FF unit to refer to the VUE and perform any required operations on it (e.g., cause it to be read into the thread payload, dereference it, etc.).
- **Primitive Topology Information**: This information is used to identify/delineate primitive topologies in the 3D pipeline. Initially, the VF unit supplies this information, which then passes thru the VS stage unchanged. GS and CLIP threads must supply this information with each vertex they produce (via the URB_WRITE message). If a FF unit directly outputs vertices (that were not generated by a thread they spawned), that FF unit is responsible for providing this information.
  - **PrimType**: The type of topology, as defined by the corresponding field of the 3DPRIMITIVE command.
  - **StartPrim**: TRUE only for the first vertex of a topology.
  - **EndPrim**: TRUE only for the last vertex of a topology.
- Debug information (refer to *Debugging* chapter)
  - The FF unit which owns the VUE
  - Sequence numbers which uniquely identify (with some limits) the VUE output by the owning FF unit. (This data can be used to trap on a specific vertex)
- (Possibly, depending on FF unit) Data read back from the **Vertex Header** of the VUE.

2.6.1 Vertex URB Entry (VUE) Formats

In general, vertex data is stored in Vertex URB Entries (VUEs) in the URB, processed by CLIP threads, and only referenced by the pipeline stages indirectly via VUE handles. Therefore (for the most part) the contents/format of the vertex data is not exposed to 3D pipeline hardware – the FF units are typically only aware of the handles and sizes of VUEs.

VUEs are written in two ways:

- At the top of the 3D Geometry pipeline, the VF's InputAssembly function creates VUEs and initializes them from data extracted from Vertex Buffers as well as internally-generated data.
- VS, GS, and CLIP threads can compute, format and write new VUEs as thread output.

There are only two points in the 3D FF pipeline where the FF units are exposed to the VUE data. Otherwise the VUE remains opaque to the 3D pipeline hardware.

- Just prior to the CLIP stage, all VUEs are read-back:
  - Readback of the Vertex Header (first 256 bits of the VUE)
- Just after the CLIP stage, on clip-generated VUEs are read-back:
  - Readback of the Vertex Header (first 256 bits of the VUE)

Software must ensure that any VUEs subject to readback by the 3D pipeline start with a valid Vertex Header. This extends to all VUEs with the following exceptions listed below:

- If the VS function is enabled, the VF-written VUEs are not required to have Vertex Headers, as the VS-incoming vertices are guaranteed to be consumed by the VS (i.e., the VS thread is responsible for overwriting the input vertex data).
- If the GS FF is enabled, neither VF-written VUEs nor VS thread-generated VUEs are required to have Vertex Headers, as the GS will consume all incoming vertices.
- (There is a pathological case where the CLIP state can be programmed to guarantee that all CLIP-incoming vertices are consumed – regardless of the data read back prior to the CLIP stage – and therefore only the CLIP thread-generated vertices would require Vertex Headers).

The folllowing table defines the Vertex Header. The Position fields are described in further detail below.

Table 2-2. VUE Vertex HeaderVUE Vertex Header

DWord	Bit	Description
D0	31:0	Reserved: MBZ
D1	31:0	Reserved: MBZ
D2	31:0	Reserved: MBZ
D3	31:19	Reserved: MBZ
	18:8	**Point Width.** This field specifies the width of POINT objects in screen-space pixels. It is used only for vertices within POINTLIST and POINTLIST_BF primitive topologies, and is ignored for vertices associated with other primitive topologies. This field is read back by both the GS and Clip units. Format: U8.3 pixels
	7:0	**User Clip Codes.** These are 'outside' status bits associated with the vertex element components marked as CullDistance or ClipDistance. The JITTER is required to generate code to compute and pack these bits. If a Cull/ClipDistance value is negative or a NaN value, its corresponding User Clip Code bit should be set. Up to eight values/bits are supported. The CLIP unit supports the **UserClipFlag ClipTest Enable Bitmask** (CLIP_STATE) which is applied to this field before being used in ClipTest. This field is read back only by the GS unit. This field is ignored for CLIP thread-generated vertices, as this information is only relevant to CLIP input vertices. Format: BITMASK8
D4	31:0	**Vertex Position X Coordinate.** If this is a PREMAPPED vertex, this field contains the X component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the X component of the vertex's NDC space position (i.e., the clip space X component divided by the clip space W component). Format: FLOAT32

DWord	Bit	Description
D5	31:0	**Vertex Position Y Coordinate.** If this is a PREMAPPED vertex, this field contains the Y component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Y component of the vertex's NDC space position (i.e., the clip space Y component divided by the clip space W component). Format: FLOAT32
D6	31:0	**Vertex Position Z Coordinate.** If this is a PREMAPPED vertex, this field contains the Z component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Z component of the vertex's NDC space position (i.e., the clip space Z component divided by the clip space W component). Format: FLOAT32
D7	31:0	**Vertex Position RHW Coordinate.** This field contains the reciprocal of the vertex's clip space W coordinate. Format: FLOAT32
(D8-Dn)	31:0	**(Remainder of Vertex Elements).** While DWords D0-D7 are exposed to the device (i.e., read back by FF units), DWords D8-Dn of vertices written (by threads) are opaque to the device. Software is free to format/use these DWords as desired. The absolute maximum size limit on this data is specified via a maximum limit on the amount of data that can be read from a VUE (including the Vertex Header) (**Vertex Entry URB Read Length** has a maximum value of 63 256-bit units). Therefore the Remainder of Vertex Elements has an absolute maximum size of 62 256-bit units. Of course the actual allocated size of the VUE can and will limit the amount of data in a VUE.

2.6.2 Vertex Positions

(For the sake of brevity, the following discussion will use the term *map* as a shorthand for "compute screen space coordinate via perspective divide followed by viewport transform".)

The "Position" fields of the Vertex Header are the only vertex position coordinates exposed to the 3D Pipeline. The CLIP and SF units are the only FF units which perform operations using these positions. The VUE will likely contain other position attributes for the vertex outside of the Vertex Header, though this information is not directly exposed to the FF units. For example, the Clip Space position will likely be required in the VUE (outside of the Vertex Header) in order to perform correct and robust 3D Clipping in the CLIP thread.

In the CLIP unit, the read-back Position fields are interpreted as being in one of two coordinate systems, depending on the **CLIP_STATE.VertexPositionSpace** bit. The CLIP unit will modify its VertexClipTest function depending on the coordinate space of the incoming vertices.

- **VPOS_NDCSPACE (Normalized Device Coordinate Space position, post-perspective division)**: This is the typical coordinate space in which vertex positions are defined upon input to the CLIP unit. A *speculative* perspective-division will have been performed, though the viewport map transformation will not have been applied (as this is provided by the downstream SF FF unit). An advantage of clip-testing in NDC space is that the View Volume has canonical unit

dimensions (i.e., it's cheap to test against). The "speculative" nature of the perspective divide is discussed below.

- **VPOS_SCREENSPACE (Screen Space position)**: Under certain circumstances, the position in the Vertex Header will contain the screen-space (pixel) coordinates (post viewport mapping).

The SF unit does not have a state bit defining the coordinate space of the incoming vertex positions. Software must use the Viewport Mapping function of the SF unit in order to ensure that screen-space coordinates are available after that function. If screen space coordinates are passed into SF, then software will likely turn off the Viewport Mapping function.

The following subsections briefly describe the three relevant coordinate spaces.

2.6.2.1 Clip Space Position

The *clip-space* position of a vertex is defined in a homogeneous 4D coordinate space where, after perspective projection (division by W), the visible "view volume" is some canonical (3D) cuboid. Typically the X/Y extents of this cuboid are [-1,+1], while the Z extents are either [-1,+1] or [0,+1]. The API's VS or GS shader program will include geometric transforms in the computation of this clip space position such that the resulting coordinate is positioned properly in relation to the view volume (i.e., it will include a "view transform" in this computation path).

Note that, under typical perspective projections, the clip-space W coordinate is equal to the view-space Z coordinate.

A vertex's clip-space coordinates must be maintained in the VUE up to 3D clipping, as this clipping is performed in clip space.

- Clip-space position are stored outside of (beyond) the Vertex Header. VS/GS/Clip kernels must perform perspective projection internally and subsequently store the post-projected (NDC-space, see below) position in the Vertex Header for use by the FF pipeline.

2.6.2.2 NDC Space Position

A perspective divide operation performed on a clip-space position yields a [X,Y,Z,RHW] NDC (Normalized Device Coordinates) space position. Here "normalized" means that visible geometry is located within the [-1,+1] or [0,+1] extent view volume cuboid (see clip-space above).

- The NDC X,Y,Z coordinates are the clip-space X,Y,Z coordinates (respectively) divided by the clip-space W coordinate (or, more correclty, the clip-space X,Y,Z coordinates are multiplied by the reciprocal of the clip space W coordinate).
  - Note that the X,Y,Z coordinates may contain INFINITY or NaN values (see below).
- The NDC RHW coordinate is the reciprocal of the clip-space W coordinate and therefore, under normal perspective projections, it is the reciprocal of the view-space Z coordinate. Note that NDC space is really a 3D coordinate space, where this RHW coordinate is retained in order to perform perspective-correct interpolation, *et al*. Note that, under typical perspective projections.
  - Note that the RHW coordinate make contain an INFINITY or NaN value (see below).

2.6.2.2.1 Speculative Perspective Divide

When operating in VPOS_NDCSPACE mode, the CLIP stage requires a 'speculative' PerspectiveDivide to have been performed on all incoming vertices. This places a requirement on software (the JITTER) to cause the NDC coordinates to be computed and stored prior to the CLIP stage, in addition to any shader functions which may be required. In the case where the application simply inputs clip space positions without any intervening processing prior to the CLIP stage, software must cause the speculative PerspectiveDivide function to be performed in the VS thread.

This PerspectiveDivide function is considered speculative in that the results may not be used, i.e., in the case where the vertex lies outside the clipping boundaries. Note that, when performing PerspectiveDivide before 3DClipping, the resulting NDC coordinates may not even be representable. For example, the clip-space W coordinate may be zero or close enough to zero to cause the X/W, Y/W or Z/W operation to result in an INFINITE value. However, in these cases, the PerspectiveDivide results will not be used, and instead the corresponding clip-space coordinates will be used as input to the 3DClipping function (assuming the object is not trivially rejected).

NaN Values in NDC Coordinate Components

There are cases where a speculative PerspectiveDivide can produce NaN results. The following table shows these cases for the computation of X/W (same holds true for Y/W and Z/W).

W	RHW	Clip X	NDC X = X*RHW	Comments
NaN	NaN	d/c	NaN	Clip space position not representable (W is NaN)
d/c	d/c	NaN	NaN	Clip space position not representable (X is NaN)
+/-INF	+/-0	+/-INF	NaN	Clip space position is representable, but 3D clipping will not yield valid results.
+/-0 or denorm	+/-INF	+/-0 or denorm	NaN	Clip space postiion is representable. This is a case where a NDC X,Y,Z component can be NaN even when the Clip space position is representable. 3D Clipping can yield valid results.
+/-INF	+/-0	Not +/-INF	+/-0	This is the case of infinite perspective, where the vertex collapses to the NDC origin.
+/-0 or denorm	+/-INF	Not (+/-0 or denorm)	+/-INF	This is a case where an infinite NDC coordinate is generated, though 3D Clipping will be able to produce valid results.

During VertexClipTest, any vertex with an NaN NDC RHW coordinate will be marked as "BAD". During ClipDetermination, any object containing a 'BAD' vertex will be trivially rejected.

2.6.2.3 Screen-Space Position

Screen-space coordinates are defined as:

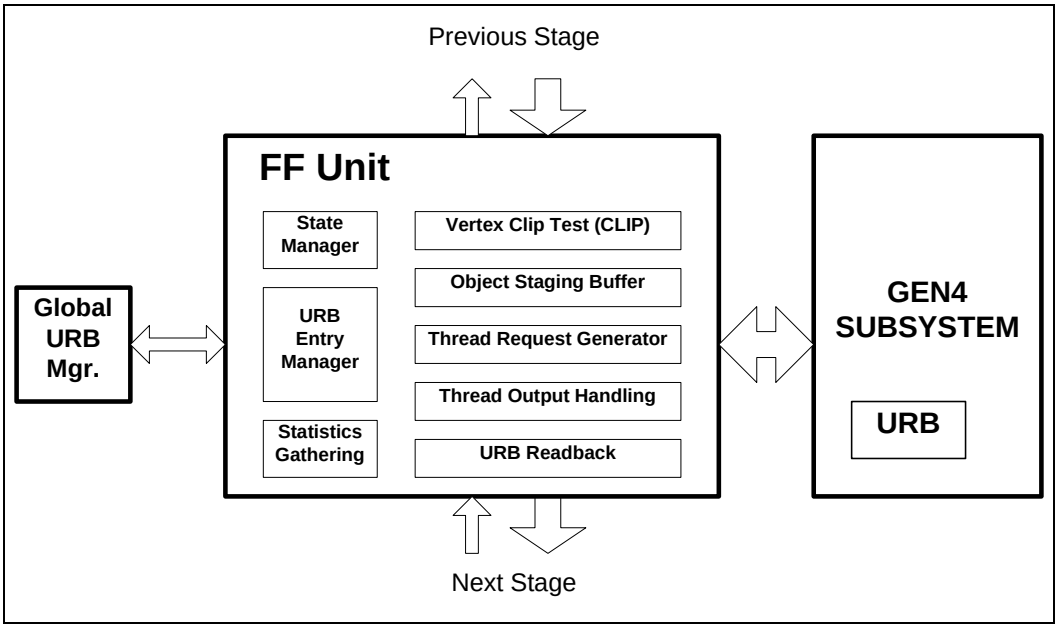
- X,Y coordinates are in absolute screen space (pixel coordinates, upper left origin). See Vertex X,Y Clamping and Quantization in the SF section for a discussion of the limitations/restrictions placed on screenspace X,Y coordinates.
- Z coordinate has been mapped into the range used for DepthTest.
  - D3D allows the visible Z range ([0,1] NDC) to be mapped into some subrange within [0,1]. However, by definition, pre-mapping in D3D disables Z clipping. (If mapped Z coordinates outside of [0,1] are presented, rendering results are undefined.) Software must explicitly disable Z clipping via **Viewport Z ClipTest Enable** (CLIP_STATE) whenever positions are pre-mapped.
- RHW coordinate is actually the reciprocal of clip-space W coordinate (typically the reciprocal of the view-space Z coordinate). D3D requires RHW to be positive, or rendering results are undefined.

2.7 3D Pipeline Stage Overview

The fixed-function (FF) stages of the 3D pipeline share some common functionality, specifically related to the creation and management of threads. This chapter is intended to describe the behavior and programming model of these common functions, in an effort to not replicate this information for each pipeline stage. Stage-specific exceptions to the information provided here will be included in the stage-specific chapters to follow.

2.7.1 Generic 3D FF Unit Block Diagram

The following block diagram, in general, applies to the VS, GS and CLIP stages.



2.7.2 Common 3D FF Unit Functions

A major role of the FF stages is in managing the GEN4 threads that perform the majority of the processing on the vertex/pixel data. (In general, the amount of non-thread processing performed by the 3DPIPE stages increases towards the end of the pipeline.) In a generic sense, the key functions included are:

- Bypass Mode
- URB Entry Management
- Thread Initiation Management
- Thread Request Data Generation
  - Thread Control Information Generation
  - Thread Payload Header Generation
  - Thread Payload Data Generation
- Thread Output Handling
- URB Entry Readback
- Statistics Gathering

The following table lists the various state variables used to control the common FF functions:

State Variable	Programmed Via	Generic Functions Affected
<stage> Enable	3DSTATE_PIPELINED_POINTERS	Bypass Mode
Kernel Start Pointer	Pipeline State Descriptor	Thread Request Data Gen.
GRF Register Block Count	Pipeline State Descriptor	Thread Request Data Gen.
Single Program Flow	Pipeline State Descriptor	Thread Request Data Gen.
Thread Priority	Pipeline State Descriptor	Thread Request Data Gen.
Floating Point Mode	Pipeline State Descriptor	Thread Request Data Gen.
Exceptions Enable	Pipeline State Descriptor	Thread Request Data Gen.
Scratch Space Base Pointer	Pipeline State Descriptor	Thread Request Data Gen.
Per Thread Scratch Space	Pipeline State Descriptor	Thread Request Data Gen.
Constant URB Entry Read Length	Pipeline State Descriptor	Payload Data Gen.
Constant URB Entry Read Offset	Pipeline State Descriptor	Payload Data Gen.
Vertex URB Entry Read Length	Pipeline State Descriptor	Payload Data Gen.
Vertex URB Entry Read Offset	Pipeline State Descriptor	Payload Data Gen.
Dispatch GRF Start Register for URB Data	Pipeline State Descriptor	Payload Data Gen.

State Variable	Programmed Via	Generic Functions Affected
Maximum Number of Threads	Pipeline State Descriptor	Thread Resource Alloc. Scratch Space Mgt.
<stage> Fence	URB_FENCE_POINTER	URB Entry Mgt.
URB Entry Allocation Size	Pipeline State Descriptor	URB Entry Mgt.
Number of URB Handles	Pipeline State Descriptor	URB Entry Mgt.
Sampler State Pointer	Pipeline State Descriptor	Payload Header Gen.
Sampler Count	Pipeline State Descriptor	Thread Request Data Gen.
<stage> Binding Table Pointer	3DSTATE_BINDING_TABLE_POINTERS	This gets routed directly to shared functions (transparent to software).
Binding Table Entry Count	Pipeline State Descriptor	Thread Request Data Gen.
Statistics Enable	Pipeline State Descriptor	Statistics Gathering

2.7.3 Pipeline Stage Input

In general, each stage of the 3D pipeline receives inputs from the previous stage. The following table summarizes these types of input and how they are handled by a stage.

Input	Operation
• Pipelined State Commands	• All stages: The stage receives the various pipelined state commands, extracts the stage-specific information (if applicable), and then forwards the command down the pipeline.
• 3D Control Operations	• All stages: The stage receives the various pipelined 3D control operations, performs any stage-specific actions (see definition below), and then forwards the operation down the pipeline. • Refer to *3D Control Operations.*
• 3D Primitives	• CS: The CS unit receives 3DPRIMITIVE commands directly from its Command Input function, and passes the relevant parameters to the VF stage. • VF: The VF stage receives 3DPRIMITIVE command information from the CS unit, executes that command, stores the resultant vertex data in the URB and passes corresponding vertex information packets down the pipeline. • VS, GS, CLIP, SF, WM: The stage receives 3D primitive topologies as sequences of vertex information packets. These vertices are the result of a 3DPRIMITIVE command, where the source vertices may have undergone processing by previous pipeline stages.

2.7.4 Pipelined State Commands

2.7.4.1 URB_FENCE

The URB_FENCE command is used to reallocate the URB amongst the various pipeline stages. The actual fence values are passed from the CS to a Global URB Manager function, and only an indication of a fence change is propagated down the pipeline. (See *Graphics Processing Engine* for a description of this command).

FF Stage	Support
VF	N/A. The VF unit uses the UEs allocated to the VS unit.
CS **VS** **GS** **CLIP** **SF** **WM**	When a FF unit detects a URB fence change, it first waits for any current tasks to complete. It then proceeds to initiate the deallocation of the UE handles it currently owns. It will then request reallocation of its new handles and proceed to use those new handles for subsequent activities. (*Implementation Note*: The deallocation of UE handles that are currently "free" (not in flight) is immediate. The deallocation of UE handles that are in flight will occur sometime after those handles are dereferenced (consumed by the pipeline). This trailing deallocation occurs while the FF unit proceeds onto new work.)

2.7.4.2 3DSTATE_PIPELINED_POINTERS

The state variables that control FF unit operations are primarily specified indirectly via the 3DSTATE_PIPELINED_POINTERS command. The table below summarizes each stage's handling of this command. Refer to the description of the 3DSTATE_PIPELINED_POINTERS command for a general description of that command. Refer to the stage-specific chapters for a definition and description of the various state blocks referenced by this command.

FF Stage	Support
CS	N/A
VF	N/A (its state is programmed directly with other pipelined state commands)
VS **GS** **CLIP** **SF** **WM**	Each stage extracts the pointer to its state block, and will start using that new state information for subsequent operations. The command is sent down the pipeline in order with all other pipeline traffic. The GS and CLIP units also extract a specific **Enable** bit from this command. If the unit is disabled, the corresponding state pointer is ignored, as the unit does not require this state while disabled.

2.7.4.3 3DSTATE_BINDING_TABLE_POINTERS

All stages that support the spawning of threads are required to pass a stage-specific **Binding Table Pointer** in the thread payload. The thread will subsequently pass this pointer in messages to several Shared Functions in order for those functions to access and process memory operands correctly.

The 3DSTATE_BINDING_TABLE_POINTERS command is used to pass these stage-specific pointers down the pipeline. The table below summarizes each stage's handling of this command. Refer to the description of the 3DSTATE_BINDING_TABLE_POINTERS command for a general description of that command, and the *Shared Functions* chapter for a description of the Binding Tables and their use.

FF Stage	Support
CS	N/A
VF	N/A (its state is programmed directly with other pipelined state commands)
VS **GS** **CLIP** **SF** **WM**	Each stage extracts its specific Binding Table Pointer, which it will start using for subsequent thread payloads. The command is sent down the pipeline in order with all other pipeline traffic.

2.7.4.4 CONSTANT_BUFFER

The CONSTANT_BUFFER command is used to read "constant" data from a memory buffer into a special Constant URB Entry (CURBE), and then pass the CURBE handle down the pipeline for optional inclusion in subsequent thread payloads. All stages that support the spawning of threads will extract the CURBE Handle from the command as it passes down the pipeline.

The table below summarizes each stage's handling of this command. Refer to the description of the CONSTANT_BUFFER command (*Graphics Processing Engine*) for a general description of that command, and below for a description of Constant URB Entries.

FF Stage	Support
CS	The CS unit executes the command, reading the data from the memory buffer into a CURBE, and then passes the handle of the CURBE down the pipeline (as a manifestation of this command).
VF	As the VF unit does not spawn threads, it simply passes this command down the pipeline.
VS **GS** **CLIP** **SF** **WM**	Each stage extracts the common **Constant URB Entry Handle**, which it will start using for subsequent thread payloads. The command is send down the pipeline in order with all other pipeline traffic.

2.7.5 Bypass Mode

For some (GS, CLIP) FF stages, if the associated **<FF> Enable** bit of the 3DSTATE_PIPELINED_POINTERS command is DISABLED, the stage goes into Bypass mode. In this mode, the incoming vertex and control packets are directly streamed to the next stage. Changes to binding table pointers and URB fences changes are still processed – this allows a single PIPELINE_STATE_POINTER command to enable the FF stage and commence normal operation.

Exceptions to this generic function are listed below. Refer to the FF stage's PRM chapter for details.

FF Stage	Exceptions
CS	Cannot be explicitly disabled; Bypass mode not supported.
VF	Cannot be explicitly disabled; Bypass mode not supported.
VS	While the VS stage cannot be explicitly disabled, the VS shading "function" can be disabled, causing the VF-generated VUEs to pass down the pipeline "unshaded".
GS	Supports Bypass mode, though always performs URB readback of vertices prior to CLIP stage
CLIP	Supports Bypass mode. Note that there are conditions underwhich the Clip stage must be enabled.
SF	Cannot be explicitly disabled; Bypass mode not supported.
WM	Cannot be explicitly disabled; Bypass mode not supported.

2.7.6 URB Entry Management

Note: See *Graphics Processing Engine* for a discussion of URB Allocation Requirements and Guidelines, as well as Command Ordering Rules.

Most FF stages can be allocated a number (possibly zero) of URB entries. These URB entries store the output of threads that the FF unit spawns. The following table lists which stages support URB Entry allocation:

FF Stage	URB Allocation
CS	Allocated entries used to pass constants to threads. (Optional)
VF	Not allocated entries – writes to entries (VUEs) allocated to VS. If VS is enabled, these are input to the VS thread. If VS is disabled, these entries will pass down the pipeline.
VS	Allocated entries (VUEs) used as vertex input-to and output-from a VS thread (if the VS Function is enabled) or to send raw VF-generated vertex data down the pipe (if the VS Function is disabled). (Required, as the VS stage cannot be DISABLED)
GS	Allocated entries (VUEs) to store vertex output from the GS threads (Only required if GS stage is ENABLED)

FF Stage	URB Allocation
CLIP	Allocated entries (VUEs) to store vertex output from the CLIP threads (Only required if CLIP stage is ENABLED)
SF	Allocated entries (PUEs) to store per-primitive setup results from SETUP threads. (Required)
WM	Not allocated any entries (does not store results in the URB).

The following table lists the state variables controlling the URB Entry Management for a 3D pipeline FF stage:

State Variable	Programmed Via
<stage> Fence	URB_FENCE_POINTER
URB Entry Allocation Size	Pipeline State Descriptor
Number of URB Entries	Pipeline State Descriptor

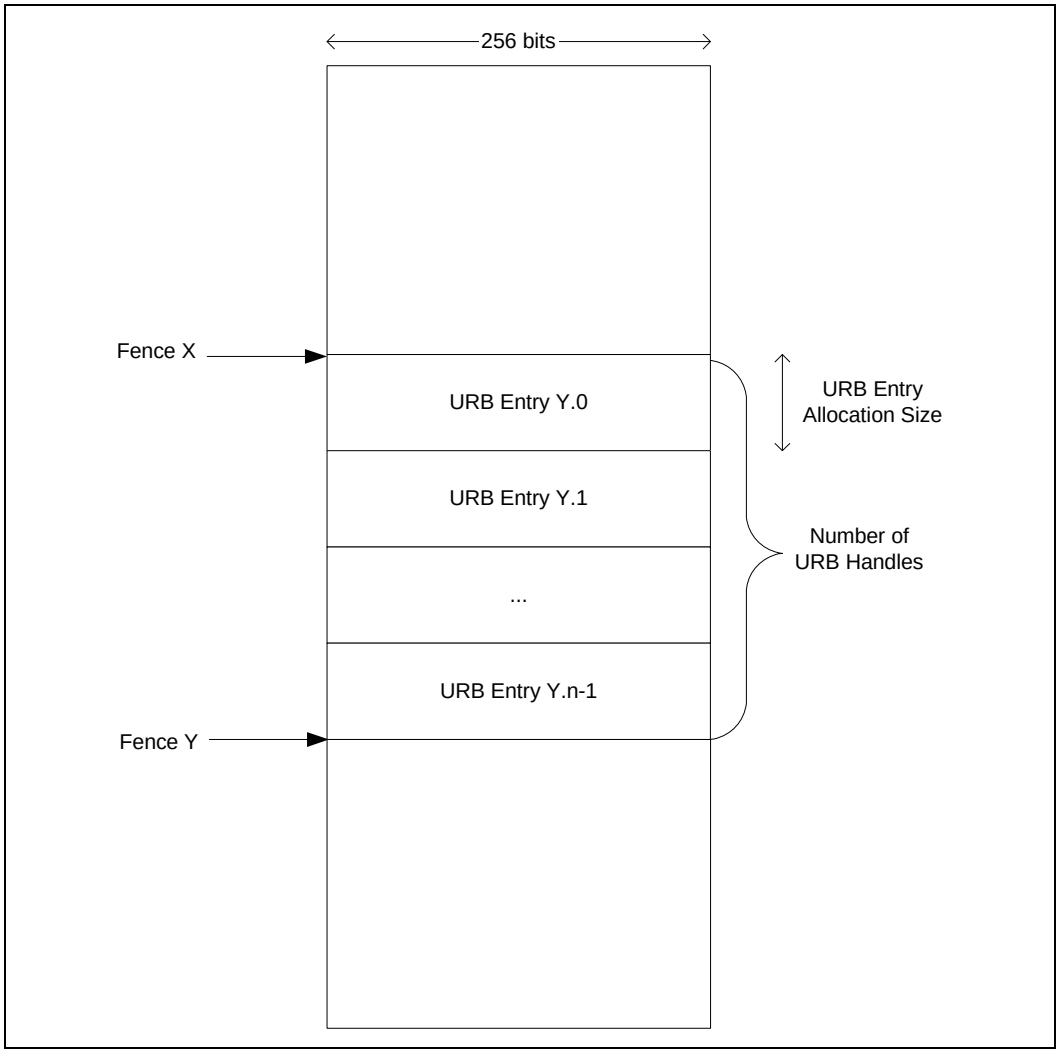
The **Number of URB Handles** state variable specifies how many URB entries are allocated for output by the FF unit.

All URB entries allocated to a particular FF stage has a size in even multiples of 256-bit URB rows (i.e, 512-bit granularity), and is specified by the **URB Entry Allocation Size** state variable. This size is for allocation purposes only. Threads may read/write less than this amount, in 256-bit units.

Where the FF stage's URB entries reside within the URB is defined via the URB_FENCE instruction. See URB Allocation (*Graphics Programming Engine*) for a description of this command.For each stage, a **<stage> Fence** state variable is specified, where the fence value for a stage specifies the ending address of that stage's allocation. Here the ending address is defined as the 512-bit row # following the stage's allocation.

If a stage does not require any URB allocation, its fence value should be set equal to the fence value of the preceding stage. The URB fence programming must accommodate the amount of URB space each enabled stage requires, i.e., (**URB Entry Allocation Size** * **Number of URB Entries**). A stage may be allocated more than this required amount, though that storage will effectively be wasted. Note that changing either the **URB Entry Allocation Size** or **Number of URB Entries** state variables requires a subsequent URB_FENCE command (see *Graphics Processing Engine* for Command Ordering Rules).

Note: One possible reason for allocating more space than required in a fenced URB region is to allow for some expansion/contraction of URB entries within that region without requiring reprogramming of adjacent regions (and the related performance impact).



2.7.7 Thread Initiation Management

Those FF stages that can spawn threads must have buffered the input (URB entries) available to supply a thread, and then ensure that there are sufficient resources (within the domain of the 3D pipeline) to make the thread request.

Once a FF stage determines a thread request can be submitted, (a) all input data required to initiate the thread is generated, (b) this information is submitted to the common thread dispatcher, (c) the thread dispatcher will spawn the thread as soon as an EU with sufficient GRF resources becomes available, and finally (d) the thread will start execution. With respect to concurrent threads, steps (c) and (d) can proceed out of order (i.e., a threads are not necessarily dispatched in the order that the thread requests are submitted to the thread dispatcher).

2.7.7.1 Thread Input Buffering

Each FF stage varies with regard to thread input requirements, and so this will not be discussed in this chapter other than the overview information provided in the following table:

FF Stage	Thread Input Requirements
CS	N/A (does not spawn threads)
VF	N/A (does not spawn threads)
VS	Normally, two vertices are buffered before a VS thread is spawned to shade the pair in parallel. Under some circumstances (e.g., a flush, state change, etc.) a single vertex will be shaded.
GS	All the vertices associated with an object must be buffered before a GS thread can be initiated to process the object.
CLIP	All the vertices associated with an object must be buffered before a CLIP thread can be initiated to process the object.
SF	All the vertices associated with an object must be buffered before a SETUP thread can be initiated to process the object.
WM	Threads spawned as required by the rasterization algorithm.

2.7.7.2 Thread Resource Allocation

Once a FF stage that spawn threads has sufficient input to initiate a thread, it must guarantee that it is safe to request the thread initiation. For all these FF stages, this check is based on :

- The **availability of output URB entries**:
  - VS: For each input URB entry, an output URB entry must be available.
  - GS: At least one output URB entry must be available to serve as the initial output vertex from the GS thread. However, software must guarantee that additional URB entries will eventually become available to allow the pipeline to make forward progress and not deadlock. There are two considerations here:
    - Single GS Threads (**Maximum Number of Threads** == 1): There must be enough GS output URB entries allocated to allow the GS thread to make progress (call this number P). P must include enough vertices to allow the next enabled stage to make progress, i.e., must contain enough vertices for the worst-case object within a primitive. For example, the system would hang if the GS stage was only allocated 2 URB entries and the GS thread tried to output a TRILIST. In this case the GS stage would need to be allocated at least 3 URB entries – the GS thread would output the first 3 vertices, then would stall on the allocation of the 4th vertex until the rest of the pipeline consumed that first triangle and dereferenced the first vertex. The clipper, when enabled, imposes additional requirements on the number of output URB entries allocated to the GS. Because of the way the clipper processes strip/fan

primitives, it will not release the URB entries for the vertices of a given object until it has finished processing the *next* object in the primitive. The minimum number of handles that must be allocated to the GS for strip/fan –type primitives is thus increased according to the following table:

Topology	Minimum GS Handles
LINESTRIP, LINESTRIP_BF, LINESTRIP_CONT, LINESTRIP_CONT_BF	**3**
POLYGON, TRIFAN, TRIFAN_NOSTIPPLE	**4**
TRISTRIP, TRISTRIP_REV	**5**

        - Concurrent GS threads: If more than one concurrent GS thread is permitted, software must account for the possibility that all subsequent GS threads complete before the preceding GS thread outputs its first vertex. Therefore, if N concurrent threads are permitted, and each GS requires P URB handles, there must be enough GS URB entries allocated to accommodate (N-1)*P entries for the subsequent threads plus P entries to ensure the preceding thread can make progress, for a total of N*P entries.
    - CLIP: Same considerations as GS (above)
    - SF: An output URB entry must be available to store the results of the SETUP thread.
    - WM: N/A (does not output to URB)
- The **Maximum Number of Threads** state variable. This state variable limits the number of concurrent threads a FF stage can have executing. As long as the FF stage is operating below this limit, it can make additional thread initiation requests.
- In addition, the WM unit utilizes a **scoreboard** mechanism to ensure proper ordering of operations – and this mechanism can postpone the initiation of new threads. (See Windower chapter).

Software is responsible for programming of **Maximum Number of Threads** to ensure the correct and optimal operation of the 3D pipeline.

The considerations for programming **Maximum Number of Threads** are summarized below:

1. **URB Allocation**: (See discussion above)

2. **Scratch Space Allocation**: When the current kernel of an enable stage requires use of scratch space (for API-defined temporary storage, register spill/fill, overflow stacks, etc.), software must limit the number of concurrent threads (via **Maximum Number of Threads**) such that the total scratch space requirement is satisfied by the amount of scratch space memory allocated to the FF stage.

3. **Stream Output Serialization**: If a kernel is required to output a serialized stream of data to a memory buffer (e.g., a GS thread supporting D3D10's Stream Output function), threads for that stage must be serialized by SW only allowing (**Maximum Number of Threads** == 1).

4. **Performance**: In general, a larger number of possibly-concurrent threads will better ensure the GEN4 cores are fully utilized.

Note: The 3D pipeline can function correctly with (**Maximum Number of Threads** == 1) set at each enabled stage, given that there are sufficient resources to run this single thread (scratch space, etc). However, this will certainly not be an optimal configuration. See *Graphics Processing Engine* for a discussion of URB Allocation Requirements and Guidelines which includes information on programming the Number Of Threads for the various FF units.

2.7.8 Thread Request Generation

Once a FF unit determines that a thread can be requested, it must gather all the information required to submit the thread request to the Thread Dispatcher. This information is divided into several categories, as listed below and subsequently described in detail.

- **Thread Control Information**: This is the information required (from the FF unit) to establish the execution environment of the thread. Note that some information affecting the thread execution state is programmed external to the 3D pipeline (e.g., Exception Handler IP, Breakpoint IP, etc.) See *Debugging* chapter.
- **Thread Payload Header**: This is the first portion of the thread payload passed in the GRF, starting at GRF R0. This is information passed directly from the FF unit. It precedes the Thread Payload Input URB Data.
- **Thread Payload Input URB Data**: This is the second portion of the thread payload. It is read from the URB using entry handles supplied by the FF unit.

2.7.8.1 Thread Control Information

The following table describes the various state variables that a FF unit uses to provide information to the Thread Dispatcher and which affect the thread execution environment. Note that this information is not directly passed to the thread in the thread payload (though some fields may be subsequently accessed by the thread via architectural registers).

Table 2-3. State Variables Included in Thread Control Information

State Variable	Usage	FFs
Kernel Start Pointer	This field, together with the **General State Pointer**, specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Pointer**.	All FFs spawning threads (VS, GS, CLIP, SF, WM)
GRF Register Block Count	Specifies, in 16-register blocks, how many GRF registers are required to run the kernel. The Thread Dispatcher will only seek candidate EUs that have a sufficient number of GRF register blocks available. Upon selecting a target EU, the Thread DIspatcher will generate a logical-to-physical GRF mapping and provide this to the target EU.	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Single Program Flow (SPF)	Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*.	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Thread Priority	The Thread Dispatcher will give priority to those thread requests with Thread Priority of HIGH_PRIORITY over those marked as LOW_PRIORITY. Within these two classes of thread requests, the Thread Dispatcher applies a priority order (e.g., round-robin --- though this algorithm is considered a device implementation-dependent detail).	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Floating Point Mode	This determines the initial value of the **Floating Point Mode** bit of the EU's CR0 architectural register that controls floating point behavior in the EU core. (See ISA.)	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Exceptions Enable	This bitmask controls the exception hanlding logic in the EU. (See ISA.)	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Sampler Count	This is a hint which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).	All stages supporting sampling (VS, GS, WM)
Binding Table Entry Count	This is a hint which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The comments included in Sampler Count (above) also apply to this field).	All FFs spawning threads (VS, GS, CLIP, SF, WM)

2.7.8.2 Thread Payload Generation

FF units are responsible for generating a thread *payload* – the data pre-loaded into the target EU's GRF registers (starting at R0) that serves as the primary direct input to a thread's kernel. The general format of these payloads follow a similar structure, though the exact payload size/content/layout is unique to each stage. This subsection describes the common aspects – refer to the specific stage's chapters for details on any differences.

The payload data is divided into two main sections: the *payload header* followed by the *payload URB data*. The payload header contains information passed directly from the FF unit, while the payload URB data is obtained from URB locations specified by the FF unit.

Note: The first 256 bits of the thread payload (the initial contents of R0, aka "the R0 header") is specially formatted to closely match (and in some cases exactly match) the first 256 bits of thread-generated *messages* (i.e., the message header) accepted by shared functions. In fact, the send instruction supports having a copy of a GR's contents (such as R0) used as the message header. Software must take this intention into account (i.e., "don't muck with R0 unless you know what you're doing"). This is especially important given the fact that several fields in the R0 header are considered opaque to SW, where use or modification of their contents might lead to UNDEFINED results.

The payload header is further (loosely) divided into a leading *fixed payload header* section and a trailing, variable-sized *extended payload header* section. In general the size, content and layout of both payload header sections are FF-specific, though many of the fixed payload header fields are common amongst the FF stages. The extended header is used by the FF unit to pass additional information specific to that FF unit. The extended header is defined to start after the fixed payload header and end at the offset defined by **Dispatch GRF Start Register for URB Data**. Software can cause use the **Dispatch GRF Start Register for URB Data** field to insert padding into the extended header in order to maintain a fixed offset for the start of the URB data.

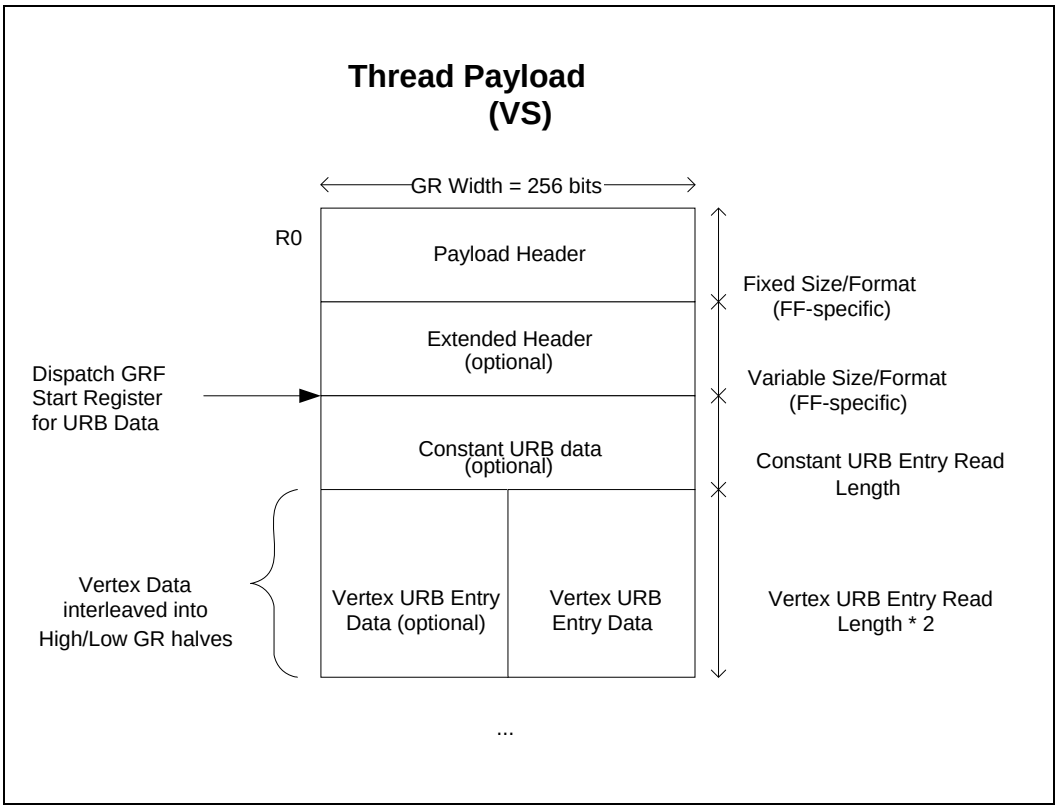
Following the payload header is the payload URB data. The FF unit provides the information (handles, etc.) used by the GEN4 subsystem to read specific portions of the URB and subsequently load this data as part of the thread payload.

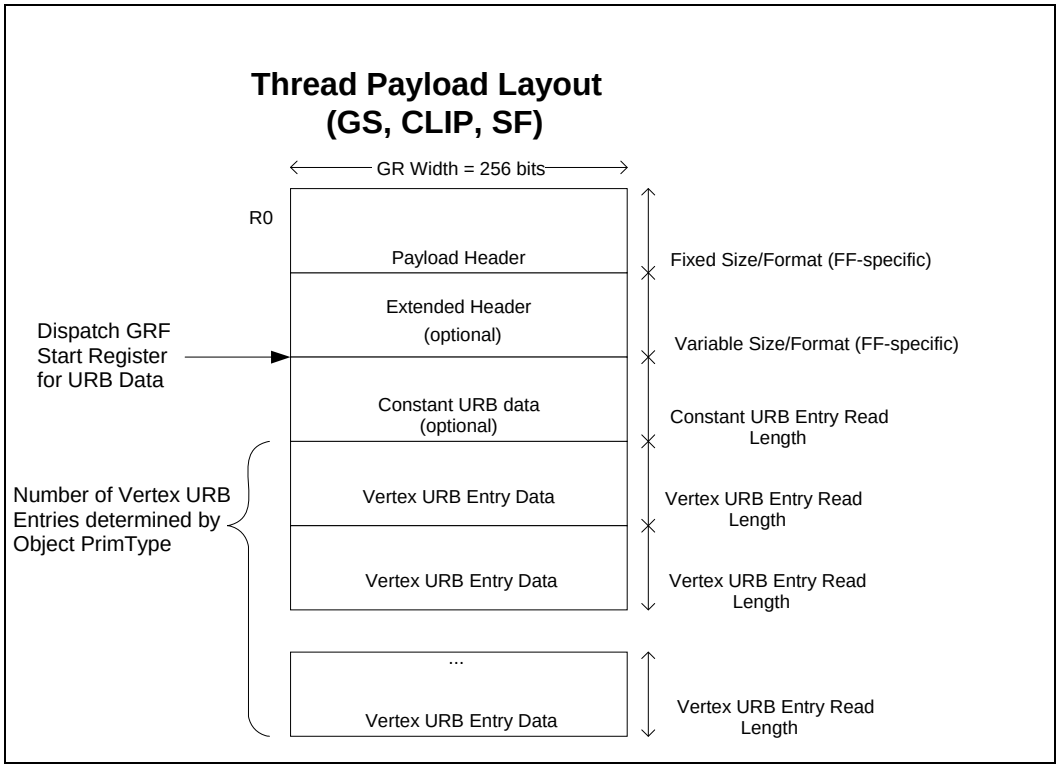
Table 2-4. Payload Sizes

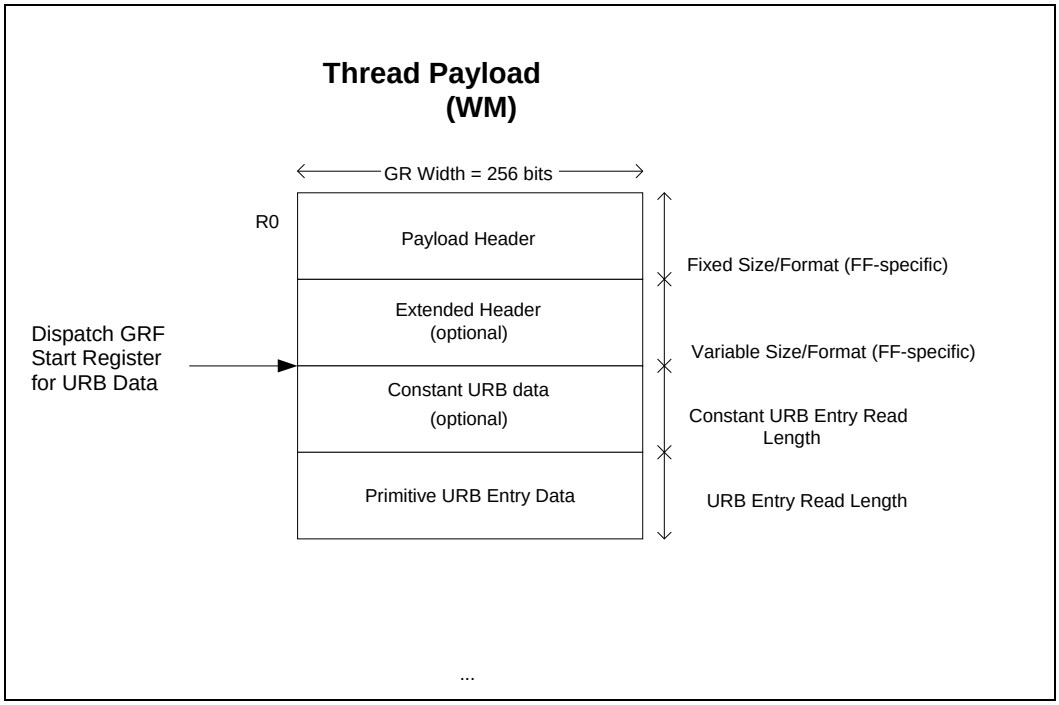
FF Stage	Fixed Payload Header Size (# GRF Regs)	Extended Payload Header Size (# GRF Regs)	URB Data
CS	N/A	N/A	N/A
VF	N/A	N/A	N/A
VS	1 (R0 only)	0 (typically, can be non-zero if padding requried)	Optional Constant URB data + 2 (interleaved) Vertex URB Entries
GS	1 (R0 only)	0 (typically, can be non-zero if padding requried)	Optional Constant URB data + 1-6 Vertex URB Entries
CLIP	1 (R0 only)	0 (typically, can be non-zero if padding requried)	Optional Constant URB data + 1-3 Vertex URB Entries

FF Stage	Fixed Payload Header Size (# GRF Regs)	Extended Payload Header Size (# GRF Regs)	URB Data
SF	3 (R0-R2)	0 (typically, can be non-zero if padding requried)	Optional Constant URB data + 1-3 Vertex URB Entries
WM	2 (R0-R1)	Variable (see WM chapter)	Optional Constant URB data + 1 Primitive URB Entries

The following diagrams show the general layout of the various thread payloads. Refer to the specific FF stage chapters for details.




Thread Payload Layout
(GS, CLIP, SF)
GR Width = 256 bits
R0
Payload Header
Fixed Size/Format (FF-specific)
Extended Header
(optional)
Variable Size/Format (FF-specific)
Dispatch GRF
Start Register
for URB Data
Constant URB data
(optional)
Constant URB Entry Read
Length
Number of Vertex URB
Entries determined by
Object PrimType
Vertex URB Entry Data
Vertex URB Entry Read
Length
Vertex URB Entry Data
Vertex URB Entry Read
Length
...
Vertex URB Entry Data
Vertex URB Entry Read
Length


Thread Payload
(WM)
GR Width = 256 bits
R0
Payload Header
Fixed Size/Format (FF-specific)
Extended Header
(optional)
Variable Size/Format (FF-specific)
Dispatch GRF
Start Register
for URB Data
Constant URB data
(optional)
Constant URB Entry Read
Length
Primitive URB Entry Data
URB Entry Read Length
...

2.7.8.2.1 Fixed Payload Header

The payload header is used to pass FF pipeline information required as thread input data. This information is a mixture of SW-provided state information (state table pointers, etc.), primitive information received by the FF unit from the FF pipeline, and parameters generated/computed by the FF unit. most of the fields of the fixed header are common between the FF stages. These non-FF-specific fields are described in Table 2-5. Note that a particular stage's header may not contain all these fields, so they are not "common" in the strictest sense.

Table 2-5. Fixed Payload Header Fields (non-FF-specific)

Fixed Payload Header Field (non-FF-specific)	Description	FFs
FF Unit ID	Function ID of the FF unit. This value identifies the FF unit within the GEN4 subsystem. The FF unit will use this field (when transmitted in a Message Header to the URB Function) to detect messages emanating from its spawned threads.	All FFs spawning threads
Snapshot Flag	Set when the FF unit detects when this thread dispatch matches certain debug criteria.	All FFs spawning threads
Thread ID	This field uniquely identifies this thread within the FF unit over some period of time. See *Debugging* chapter.	All FFs spawning threads
Scratch Space Pointer	This is the starting location of the thread's allocated scratch space, specified as an offset from the **General State Base Address**. Note that scratch space is allocated by the FF unit on a per-thread basis, based on the **Scratch Space Base Pointer** and **Per-Thread Scratch Space Size** state variables. FF units will assign a thread an arbitrarily-positioned region within this space. The scratch space for multiple (API-visible) entities (vertices, pixels) will be interleaved within the thread's scratch space.	All FFs spawning threads
Dispatch ID	This field identifies this thread within the outstanding threads spawned by the FF unit. This field does not uniquely identify the thread over any significant period of time. *Implementation Note*: This field is effectively an "active thread index". It is used on a thread's URB allocation request to identify which thread's handle pool is to source the allocation. It is used upon thread termination to free up the thread's scratch space allocation.	All FFs spawning threads

Fixed Payload Header Field (non-FF-specific)	Description	FFs
Binding Table Pointer	This field, together with the **Surface State Base Pointer**, specifies the starting location of the Binding Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **Surface State Base Pointer**. See *Shared Functions* for a description of a Binding Table.	All FFs spawning threads
Sampler State Pointer	This field, together with the **General State Base Pointer**, specifies the starting location of the Sampler State Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **General State Base Pointer**. See *Shared Functions* for a description of a Sampler State Table.	All FFs spawning threads which sample (VS, GS, WM)
Per Thread Scratch Space	This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.	All FFs spawning threads
Handle ID <n>	This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS,GS,CLIP, SF
URB Return Handle <n>	This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. hreads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS,GS,CLIP, SF

Fixed Payload Header Field (non-FF-specific)	Description	FFs
Primitive Topology Type	As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (e.g., for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread. GEN4 kernels written to process different types of objects can use this value to direct that processing. E.g., when a CLIP kernel is to provide clipping for all the various primitive types, the kernel would need to examine the Primitive Topology Type to distinguish between point, lines, and triangle clipping requests. NOTE: In general, this field is identical to the Primitive Topology Type assoociated with the primitive vertices as received by the FF unit. Refer to the individual FF unit chapters for cases where the FF unit modifies the value before passing it to the thread. (E.g., certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).	GS, CLIP, SF, WM

2.7.8.2.2 Extended Payload Header

The extended header is of variable-size, where inclusion of a field is determined by FF unit state programming. Only the WM stage supports extended headers. Refer to the Windower (WM) chapter for the size/content/layout of the extended headers.

In order to permit the use of common kernels (thus reducing the number of kernels required), the **Dispatch GRF Start Register for URB Data** state variable is supported in all FF stages. This SV is used to place the payload URB data at a specific starting GRF register, irrespective of the size of the extended header. A kernel can therefore reference the payload URB data at fixed GRF locations, while conditionally referencing extended payload header information.

2.7.8.2.3 Payload URB Data

In each thread payload, following the payload header, is some amount of URB-sourced data required as input to the thread. This data is divided into an optional *Constant URB Entry* (CURBE), following either by a Primitive URB Entry (WM) or a number of Vertex URB Entries (VS, GS, CLIP, SF). A FF unit only knows the location of this data in the URB, and is never exposed to the contents. For each URB entry, the FF unit will supply a sequence of handles, read offsets and read lengths to the GEN4 subsystem. The subsystem will read the appropriate 256-bit locations of the URB, optionally perform swizzling (VS only), and write the results into sequential GRF registers (starting at **Dispatch GRF Start Register for URB Data**).

Table 2-6. State Variables Controlling Payload URB Data

State Variable	Usage	FFs
Dispatch GRF Start Register for URB Data	This SV identifies the starting GRF register receiving payload URB data. Software is responsible for ensuring that URB data does not overwrite the Fixed or Extended Header portions of the payload.	VS, GS, CLIP, SF, WM
Constant URB Entry Read Offset	This SV determines the starting offset with the CURBE from which constant URB data to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into the current CURBE. As the CURBE is (optionallly) used by all pipeline stages to supply constant data, this SV is used by SW to select the constants to be used for a particular stage. The sources of constant data within the CURBE for different stages can overlap. Specifying a constant data source extending beyond the end of the CURBE is UNDEFINED.	VS, GS, CLIP, SF, WM
Constant URB Entry Read Length	This SV determines the amount of data (starting from **Constant URB Entry Read Offset**) to be read from the CURBE and passed into the payload URB data. It is specified in 256-bit units. If zero, no constant data is read. SW must program a zero value whenever the Constant Buffer is invalid (i.e., the CURBE is unspecified). Specifying a constant data source extending beyond the end of the CURBE is UNDEFINED.	VS, GS, CLIP, SF, WM
Vertex URB Entry Read Offset	This SV specifies the starting offset within VUEs from which vertex data is to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into any and all VUEs passed in the payload. This SV can be used to skip over leading data in VUEs that is not required by the stage's threads (e.g., skipping over the Vertex Header data at the SF stage, as that information is not required for setup calculations). Skipping over irrelevant data can only help to improve performance. Specifying a vertex data source extending beyond the end of a vertex entry is UNDEFINED.	VS, GS, CLIP, SF
Vertex URB Entry Read Length	This SV determines the amount of vertex data (starting at **Vertex URB Entry Read Offset**) to be read from each VUEs and passed into the payload URB data. It is specified in 256-bit units. A zero value is INVALID (at very least one 256-bit unit must be read). Specifying a vertex data source extending beyond the end of a VUE is UNDEFINED.	VS, GS, CLIP, SF

Programming Restrictions: (others may already been mentioned)

- The maximum size payload for any thread is limited by the number of GRF registers available to the thread, as determined by min(128, 16 * **GRF Register Block Count**). Software is responsible for ensuring this maximum size is not exceeded, taking into account:
  - The size of the Fixed and Extended Payload Header associated with the FF unit.
  - The **Dispatch GRF Start Register for URB** Data SV.
  - The amount of CURBE data included (via **Constant URB Entry Read Length**)
  - The number of VUEs included (as a function of FF unit, it's state programming, and incoming primitive types)
  - The amount of VUE data included for each vertex (via **Vertex URB Entry Read Length**)
  - (For WM-spawned PS threads) The amount of Primitive URB Entry data.
- For any type of URB Entry reads:
  - Specifying a source region (via Read Offset, Read Length) that goes past the end of the URB Entry allocation is illegal.
    - The allocated size of Vertex/Primitive URB Entries is determined by the **URB Entry Allocation Size** value provided in the pipeline state descriptor of the FF unit owning the VUE/PUE.
    - The allocated size of CURBE entries is determined by the **URB Entry Allocation Size** value provided in the CS_URB_STATE command.

2.7.9 Thread Output Handling

Those FF units spawning threads (VS, GS, CLIP, SF, WM) are responsible for monitoring and responding to certain events generated by their spawned threads. Such events are indirectly detected by these FF units monitoring messages sent from threads to the URB Shared Function. By snooping the Message Bus Sideband and Header information, a FF can detect when a particular spawned thread sends a message to the URB function. A subset of this information is then captured and acted upon. Refer to the *URB* chapter for more details (including a table of valid/invalid combinations of the **Complete**, **Used**, **Allocate**, and **EOT** bits)

The following subsections describe functions that FF units perform as part of Thread Output Handling.

2.7.9.1 URB Entry Output (VS, GS, CLIP, SF)

(The following description is applicable only to the VS, GS, CLIP and SF stages.)

For VS, GS, CLIP, and SF threads the main (if not only) output of the thread takes the form of data written to one or more destination VUEs. At very least this is the only form of thread output visible to the FF units.

When a thread sends a URB_WRITE message to the URB function with the **Complete** and **Used** bits set in the Message Description, the spawning FF unit recognizes this as the thread having completely written a destination UE. (In the typical case of a VS thread, a pair of UEs will be written in parallel). The thread must not target any additional URB messages to this UE (unless it gets reallocated to the thread). The FF unit marks this UE as complete and available for output.

In the case where multiple concurrent threads are supported at a given stage, the FF unit is responsible for outputing UEs down the pipeline in order. I.e., all VUE outputs of a spawned thread must be sent down the pipeline (in order of allocation to the thread) prior to any outputs from a subsequently-spawned thread. This is required even if the subsequent threads perform any/all of their output prior to the preceding thread producing any/some output.

2.7.9.2 VUE Allocation (GS, CLIP)

(The following description is applicable only to the GS, CLIP stages.)

The GS and CLIP threads are passed a single, initial destination VUE handle. These threads may be required to output more than one destination VUE, and therefore they are provided with a mechanism to request additional handles.

When a GS or CLIP thread issues a URB_WRITE message with the **Allocate** bit set, the spawning FF unit will consider this a request for the allocation of an additional VUE handle. The thread must specify a destination GRF register for the message writeback data. The spawning FF unit will perform the allocation, and provide the writeback data (containing **Handle ID** and **URB Return Handle**) to the GEN4 subsystem, which will in turn deliver that data to the appropriate GRF register. (See the *URB* chapter for the definition of this writeback data).

The thread is allowed to proceed while the allocation is taking place (it is guaranteed to complete at some point). If the thread attempts to reference the writeback data before the allocation has completed, execution will be stalled in the same fashion any unfulfilled dependency is handled. It is therefore recommended that SW (a) request the additional allocation as soon as possible, and (b) reference the writeback data as late as possible in order to keep the thread in a runnable state. (Refer to the following subsection to see how the thread is allowed to “allocate ahead” and give back unused VUE handles).

Note: GS and CLIP threads must write VUEs in the order they are allocated by the FF unit (in response to an allocation request from the thread), starting with the initial destination handle passed in the thread payload.

A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a “retained” handle refers to a URB handle that (a) has been pre-allocated or allocated

and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

2.7.9.3 VUE Dereference (GS, CLIP)

(The following description is applicable only to the GS, CLIP stages.)

It is possible and legal for a GS or CLIP thread to produce no output or subsequently allocate a destination VUE that was not required (e.g., the thread allocated ahead). Therefore, there is a mechanism by which a GS/CLIP thread can "give back" (dereference) an allocated VUE. This mechanism must be used if the VUE is not written before the thread terminates.

A GS/CLIP kernel can explicitly dereference a VUE by issuing a URB_WRITE message (specifying the to-be-dereference handle) with the **Complete** bit set and the **Used** bit clear.

2.7.9.4 Thread Termination

All threads must explicitly terminate by executing a SEND instruction with the EOT bit set. (See *EU* chapters). When a thread spawned by a 3D FF unit terminates, the spawning FF unit detects this termination as a part of Thread Management. This allows the FF units to manage the number of concurrent threads it has spawned and also manage the resources (e.g., scratch space) allocated to those threads.

Programming Note: GS and Clip threads must terminate by sending a URB_WRITE message (with EOT set) with the Complete bit also set (therein returning a URB handle marked as either used or un-used).

2.7.10 VUE Readback

Starting with the CLIP stage, the 3D pipeline requires vertex information in addition to the VUE handle. For example, the CLIP unit's VertexClipTest function needs the vertex position, as does the SF unit's functions. This information is obtained by the 3D pipeline reading a portion of each vertex's VUE data directly from the URB. This readback (effectively) occurs immediately before the CLIP VertexClipTest function, and immediately after a CLIP thread completes the output of a destination VUE.

The Vertex Header (first 256 bits) of the VUE data is read back. (See the previous *VUE Formats* subsection (above) for details on the content and format of the Vertex Header.)

This readback occurs automatically and is not under software control. The only software implication is that the Vertex Header must be valid at the readback points, and therefore must have been previously loaded or written by a thread.

2.8 Synchronization of the 3D Pipeline

Two types of synchronization are supported for the 3D pipe: end-of-pipe and write synchronization. These are used to implement read and write fences as well as to write out certain statistics deterministically with respect to progress of primitives through the pipeline (and without requiring the pipeline to be flushed.) The PIPE_CONTROL command (see details below) is used to perform this synchronization.

2.8.1 End-of-Pipe Synchronization

The driver can use end-of-pipe synchronization to know that rendering is complete (although not necessarily in memory) so that it can de-allocate in-memory rendering state, read-only surfaces, instructions, and constant buffers. An end-of-pipe synchronization point is also sufficient to guarantee that all pending depth tests have completed so that the visible pixel count is complete prior to storing it to memory. End-of-pipe completion is sufficient (although not necessary) to guarantee that read events are complete (a “read fence” completion). Read events are still pending if work in the pipeline requires any type of read except a render target read (blend) to complete.

2.8.2 Write Synchronization

Write synchronization is a superset of end-of-pipe synchronization that requires that the render cache itself is flushed to memory, where the data will become globally visible. This type of synchronization is required prior to SW (CPU) actually reading the result data from memory, or initiating an operation that will use as a read surface (such as a texture surface) a previous render target.

2.8.3 Synchronization Actions

In order for the driver to act based on a synchronization point (usually the whole point), the reaching of the synchronization point must be communicated to the driver. This section describes the actions that may be taken upon completion of a synchronization point which can achieve this communication.

2.8.3.1 Writing a Value to Memory

The most common action to perform upon reaching a synchronization point is to write a value out to memory. An immediate value (included with the synchronization command) may be written. In lieu of an immediate value, the 64-bit value of the PS_DEPTH_COUNT (visible pixel count) or TIMESTAMP register may be written out to memory. The captured value will be the value at the moment all primitives parsed prior to the synchronization commands have been completely rendered, and optionally after all said primitives have been pushed to memory. It is not required that a value be written to memory by the synchronization command.

Visible pixel or TIMESTAMP information is only useful as a delta between 2 values, because these counters are free-running and are not to be reset except at initialization. To obtain the delta, two PIPE_CONTROL commands should be initiated

with the command sequence to be measured between them. The resulting pair of values in memory can then be subtracted to obtain a meaningful statistic about the command sequence.

2.8.3.1.1 PS_DEPTH_COUNT

If the selected operation is to write the visible pixel count (PS_DEPTH_COUNT register), the synchronization command should include the **Depth Stall Enable** parameter. There is more than one point at which the global visible pixel count can be affected by the pipeline; once the synchronization command reaches the first point at which the count can be affected, any primitives following it are stalled at that point in the pipeline. This prevents the subsequent primitives from affecting the visible pixel count until all primitives preceding the synchronization point reach the end of the pipeline, the visible pixel count is accurate and the synchronization is completed. This stall has a minor effect on performance and should only be used in order to obtain accurate "visible pixel" counts for a sequence of primitives.

The PS_DEPTH_COUNT count can be used to implement an (API/DDI) "Occlusion Query" function.

2.8.3.2 Generating an Interrupt

The synchronization command may indicate that a "Sync Completion" interrupt is to be generated (if enabled by the MI Interrupt Control Registers – see *Memory Interface Registers*) once the rendering of all prior primitives is complete. Again, the completion of rendering can be considered to be when the internal render cache has been updated, or when the cache contents are visible in memory, as selected by the command options.

2.8.3.3 Invalidating of Caches

If software wishes to use the notification that a synchronization point has been reached in order to reuse referenced structures (surfaces, state, or instructions), it is not sufficient just to make sure rendering is complete. If additional primitives are initiated after new data is laid over the top of old in memory following a synchronization point, it is possible that stale cached data will be referenced for the subsequent rendering operation. In order to avoid this, the PIPE_CONTROL command must be used. (See PIPE_CONTROL description below).

2.8.4 PIPE_CONTROL Command

The PIPE_CONTROL command is used to effect the synchronization described above. Parsing of a PIPE_CONTROL command does not stall the 3D pipe. Commands after PIPE_CONTROL will continue to be parsed and processed in the 3D pipeline. This may include additional PIPE_CONTROL commands. The implementation does enforce a practical upper limit (4) on the number of PIPE_CONTROL commands that may be outstanding at once. Parsing of a PIPE_CONTROL command that causes this limit to be reached will stall the parsing of new commands until the first of the outstanding PIPE_CONTROL commands reaches the end of the pipe and retires.

Note that although PIPE_CONTROL is intended for use with the 3D pipe, it *is* legal to issue PIPE_CONTROL when the Media pipe is selected. In this case PIPE_CONTROL will stall at the top of the pipe until the Media FFs finish processing commands parsed before PIPE_CONTROL. Post-synchronization operations, flushing of caches and

interrupts will then occur if enabled via PIPE_CONTROL parameters. Due to this stalling behavior, only one PIPE_CONTROL command can be outstanding at a time on the Media pipe.

PIPE_CONTROL will invalidate the Sampler and constant read caches unless the **Depth Stall Enable** bit is set. It will invalidate the Instruction/State cache if the **Instruction/State Cache Flush Enable** is set. Once notification is observed, new data may then be loaded (potentially "on top of" the old data) without fear of stale cache data being referenced for subsequent rendering.

If software wishes to access the rendered data in memory (for analysis by the application or to copy it to a new location to use as a texture, for examples), it must also ensure that the write cache (render cache) is flushed after the synchronization point is reached so that memory will be updated. This can be accomplished by setting the **Write Cache Flush Enable** bit. Note that the **Depth Stall Enable** bit must be clear in order for the flush of the render cache to occur. **Depth Stall Enable** is intended only for accurate reporting of the PS_DEPTH counter; the render cache cannot be flushed nor can the read caches be invalidated (except for the instruction/state cache) in conjunction with this operation.

Both of the vertex caches will be flushed at the end of any PIPE_CONTROL operation regardless of how the control bits are set. Note that the index-based vertex cache is always flushed between primitive topologies and of course PIPE_CONTROL can only be issued between primitive topologies. Therefore only the VF ("address-based") cache is uniquely affected by PIPE_CONTROL.

Table 2-7. Caches Invalidated/Flushed by PIPE_CONTROL Bit Settings

Depth Stall Enable	Write Cache Flush Enable	Inst/State Cache Flush Enable	Read (Sampler/Constant) Caches Inv'ed?	Write (Render) Cache Flushed?	Inst/State Cache Inv'ed?	Index-Based Vertex Cache Inv'ed?	VF Cache Inv'ed?	Stall Next Prim at Depth Stage?
0	0	0	Yes	No	No	Yes	Yes	No
0	0	1	Yes	No	Yes	Yes	Yes	No
0	1	0	Yes	Yes	No	Yes	Yes	No
0	1	1	Yes	Yes	Yes	Yes	Yes	No
1	X	0	No	No	No	Yes	Yes	Yes
1	X	1	No	No	Yes	Yes	Yes	Yes

PIPE_CONTROL

Project:	All	Length Bias:	2

The PIPE_CONTROL command is used to effect the synchronization described above.

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: 2h PIPE_CONTROL Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 00h Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	15:14	**Post-Sync Operation** Project: All This field specifies an optional action to be taken upon completion of the synchronization operation.

Value	Name	Description	Project
0h		No write occurs as a result of this instruction. This can be used to implement a “trap” operation, etc.	All
1h		Write the QWord containing Immediate Data Low, High DWs to the Destination Address	All
2h		Write the 64-bit PS_DEPTH_COUNT register to the Destination Address	All
3h		Write the 64-bit TIMESTAMP register to the Destination Address	All

Errata	Description	Project
#	PS_DEPTH_COUNT cannot be accurately sampled using this command. Setting this field to 2 will write an UNDEFINED value rather than the accurate PS_DEPTH_COUNT.	BW-A,B

PIPE_CONTROL		
	13	**Depth Stall Enable** Project: All Format: Enable If ENABLED, the 3D pipeline will stall any subsequent primitives at the Depth Test stage until the Sync and Post-Sync operations complete. If DISABLED, the 3D pipeline will not stall subsequent primitives at the Depth Test stage. This bit should be set when obtaining a "visible pixel" count to preclude the possible inclusion in the PS_DEPTH_COUNT value written to memory of some fraction of pixels from objects initiated *after* the PIPE_CONTROL command. **Programming Notes:** • This bit should be DISABLED for operations other than writing PS_DEPTH_COUNT. • This bit will have no effect (besides preventing write cache flush) if set in a PIPE_CONTROL command issued to the Media pipe.
	12	**Write Cache Flush Enable** Project: All Format: Enable Setting this bit will force Render Cache to be flushed to memory prior to this synchronization point completing. This bit should be set for all write fence Sync operations to assure that results from operations initiated prior to this command are visible in memory once software observes this synchronization. This bit should be DISABLED for End-of-pipe (Read) fences, PS_DEPTH_COUNT or TIMESTAMP queries. This bit is *ignored* if **Depth Stall Enable** is set; the Render Cache will not be flushed even if **Write Cache Flush Enable** is set.
	11	**Instruction/State Cache Flush Enable** Project: All Format: Enable Setting this bit is independent of any other bit in this packet. This bit controls the invalidation of the L1 and L2 instruction/state caches after the completion of the flush.
	10	**Reserved** Project: All Format: MBZ
	9	**Reserved** Project: All Format: MBZ
	8	**Notify Enable** Project: All Format: Enable If ENABLED, a Sync Completion Interrupt will be generated (if enabled by the MI Interrupt Control registers) once the sync operation is complete. See Interrupt Control Registers in *Memory Interface Registers* for details.
	7:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:3	**Destination Address** Project: All Address: GraphicsAddress[31:3] QW-aligned graphics memory address at which data will be written when sync point occurs. Ignored if **Post-Sync Operation** is "No write".

PIPE_CONTROL

	2	**Destination Address Type** Project: All Defines address space of Destination Address. **Value** **Name** **Description** **Project** 0h Use process local PGTT All 1h Use Global GTT; valid only for privileged commands All **Programming Notes** **Project** Ignored if “No write” is the selected in **Operation**. All
	1:0	**Reserved** Project: All Format: MBZ
2	31:0	**Immediate Data Low DW** Project: All Format: U32 Low DW of QW value to write to memory at synchronization point. Ignored if **Post-Sync Operation** is “No write”, “Write PS_DEPTH_COUNT” or “Write TIMESTAMP”.
3	31:0	**Immediate Data High DW** Project: All Format: U32 High DW of QW value to write to memory at synchronization point. Ignored if **Post-Sync Operation** is “No write”, “Write PS_DEPTH_COUNT” or “Write TIMESTAMP”.

3 Vertex Fetch (VF) Stage

3.1 Vertex Fetch (VF) Stage Overview

The VF stage performs one major function: executing 3DPRIMITIVE commands. This is handled by the VF's InputAssembly function. The InputAssembly process is closely matched to the Input Assembly function described in the D3D10 specification. Minor enhancements have been included to better support legacy D3D APIs as well as OpenGL. Refer to the D3D10 Specification for additional information, including expected usage models.

The following subsections describe some high-level concepts associated with the VF stage.

3.1.1 Input Assembly

The VF's InputAssembly function includes (for each vertex generated):

- Generation of VertexIndex for each vertex, possibly via use of an Index Buffer.
- Lookup of the VertexIndex in the Vertex Cache (if enabled)
- If a cache miss is detected:
  - Use of computed indices to fetch data from memory-resident vertex buffers
  - Format conversion of the fetched vertex data
  - Assembly of the format conversion results (and possibly some internally generated data) to form the complete "input" (raw) vertex
  - Storing the input vertex data in a Vertex URB Entry (VUE) in the URB
  - Output of the VUE handle of the input vertex to the VS stage
- If a cache hit is detected, the VUE handle from the Vertex Cache is passed to the VS stage (marked as a cache hit to prevent any VS processing).

3.1.1.1 Vertex Assembly

The VF utilizes a number of VERTEX_ELEMENT state structures to define the contents and format of the vertex data to be stored in Vertex URB Entries (VUEs) in the URB. See below for a detailed description of the command used to define these structures (3DSTATE_VERTEX_ELEMENTS).

Each active VERTEX_ELEMENT structure defines up to 4 contiguous DWords of VUE data, where each DWord is considered a "component" of the vertex element. The starting destination DWord offset of the vertex element in the VUE is specified, and the VERTEX_ELEMENT structures must be defined with monotonically increasing VUE offsets. For each component, the source of the component is specified. The source may be a constant (0, 0x1, or 1.0f), or a component of a structure in memory (e.g,. the Y component of an XYZW position in memory). In the case of a memory source, the Vertex Buffer sourcing the data, and the location and format of the source data with that VB are specified.

The VF's Vertex Assembly process can be envisioned as the VF unit stepping through the VERTEX_ELEMENT structures in order, fetching and format-converting the source information (if memory resident), and storing the results in the destination VUE.

3.1.2 Vertex Cache

The VF stage communicates with the VS stage in order to implement a Vertex Cache function in the 3D pipeline. The Vertex Cache is strictly a performance-enhancing feature and has no impact on 3D pipeline results (other than a few statistics counters).

The Vertex Cache contains the VUE handles of VS-output (shaded) vertices if the VS function is enabled, and the VUE handles of VF-output (raw) vertices if the VS function is disabled. (Note that the actual vertex data is held in the URB, and only the handles of the vertices are stored in the cache). In either case, the contents of the cache (VUE handles) are tagged with the `VertexIndex` value used to fetch the input vertex data. The rationale for using the `VertexIndex` as the tag is that (assuming no other state or parameters change) a vertex with the same `VertexIndex` as a previous vertex will have the same input data, and therefore the same result from the VF+VS function.

Note that any change to the state controlling the InputAssembly function (e.g., vertex buffer definition), or any change to the state controlling the VS function (if enabled) (e.g., VS kernel), will result in the Vertex Cache being invalidated. See Vertex Caching in *Vertex Shader* for more information on the Vertex Cache (e.g., when it is implicitly disabled, etc.)

3.2 VF Stage Input

As a stage of the GEN4 3D pipeline, the VF stage receives inputs from the previous (CS) stage.

The VF stage gets its state programmed directly via pipelined state commands. It does not support indirect state descriptors.

The following table lists the 3D pipeline commands processed by the VF stage. Other commands (not listed) are simply passed down the pipeline. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage.

Command	Description
Processing Commands	
3DPRIMITIVE	This primitive command is used to inject primitives into the 3D pipeline, where they will be processed according to the current context state settings. Most typically this processing will result in rendering to destination surfaces, though this is not required. This command is defined in the *VF Stage* chapter (as it is executed there), though the processing of this command includes the entire 3D pipeline.

Command	Description
Pipelined State Commands	
3DSTATE_INDEX_BUFFER	This pipelined state command is used to specify Index Buffer parameters used in the VF unit's InputAssembly function. An Index Buffer can be used to provide vertex indices when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See *Index Buffer* below for details on this command.
3DSTATE_VERTEX_BUFFERS	This pipelined state command is used to specify Vertex Buffer parameters used in the VF unit's InputAssembly function. Vertex Buffers provide vertex data when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See *Vertex Buffers* below for details on this command.
3DSTATE_VERTEX_ELEMENTS	This pipelined state command is used to specify Vertex Element parameters used in the VF unit's InputAssembly function. Vertex Element parameters specify how vertex data, extracted from Vertex Buffers, are format converted and stored in VUEs. This command does not travel past the VF stage. See *Input Vertex Data* below for details on this command.
3DSTATE_VF_STATISTICS	This pipelined state command is used to turn pipeline statistics gathering by the VF stage on or off. This command does not travel past the VF stage. See ***Error! Reference source not found.*** below for details on this command.

The VF stage also receives input directly from memory, in the form of Index Buffers and Vertex Buffers.

3.3 Index Buffer (IB)

The 3DSTATE_INDEX_BUFFER command is used to define an *Index Buffer* (IB) used in subsequent 3DPRIMITIVE commands.

The RANDOM access mode of the 3DPRIMITIVE command involves the use of a memory-resident IB. The IB, defined via the 3DSTATE_INDEX_BUFFER command described below, contains a 1D array of 8, 16 or 32-bit index values. These index values will be fetched by the InputAssembly function, and subsequently used to compute locations in VERTEXDATA buffers from which the actual vertex data is to be fetched. (This is opposed to the SEQUENTIAL access mode were the vertex data is simply fetched sequentially from the buffers).

Software is responsible for ensuring that accesses outside the IB do not occur. This is possible as software can compute the range of IB values referenced by a 3DPRIMITIVE command (knowing the **StartVertexLocation**, and **Vertices** values) and can then compare this range to the IB extent.

3.3.1 3DSTATE_INDEX_BUFFER

3DSTATE_INDEX_BUFFER

Project:	All	**Length Bias:**	2

This command is used to specify the current IB state used by the VF function. At most one IB is defined and active at any given time.

NOTES:

- The IB must be specified before any RANDOM 3D_PRIMITIVE commands are issued

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: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 0Ah 3DSTATE_INDEX_BUFFER Format: OpCode
	15:11	**Reserved** Project: All Format: MBZ

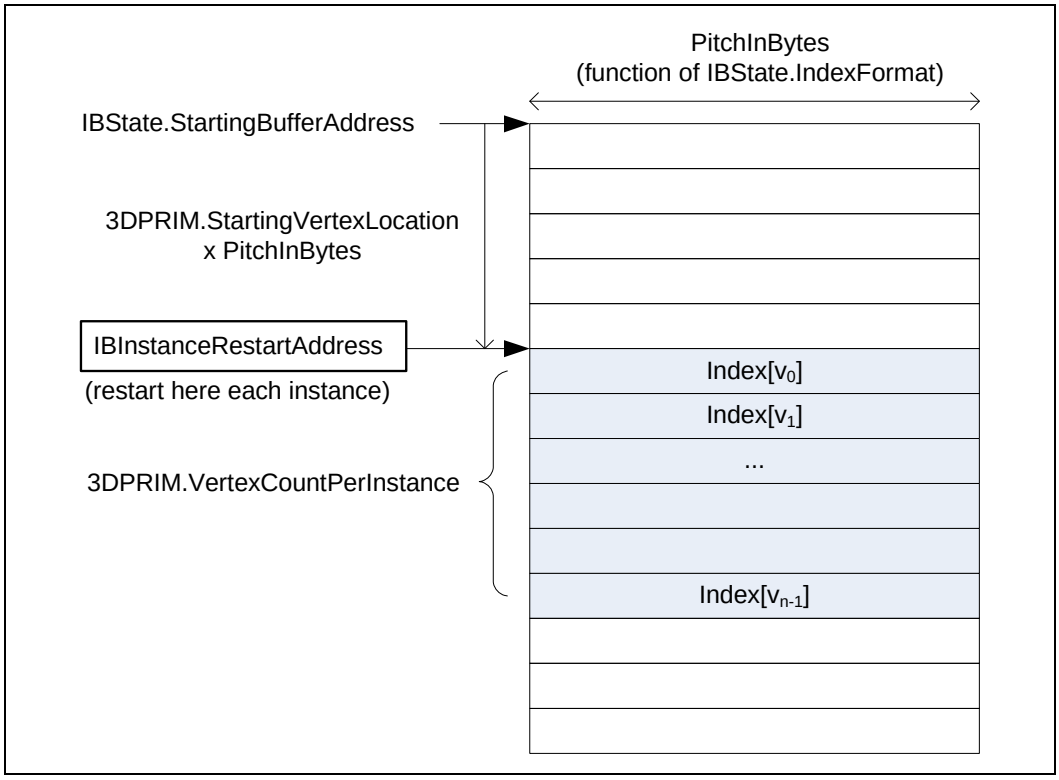
3DSTATE_INDEX_BUFFER		
2	31:0	**Buffer Ending Address** Project: All Format: GraphicsAddress[31:0] FormatDesc This field contains the address of the last valid byte in the index buffer. Any index buffer reads past this address returns 0 for all vertex elements (as if the buffer was zero-extended). Software must guarantee that the buffer ends on an index boundary (e.g., for an INDEX_DWORD buffer, Bits [1:0] == 11b)

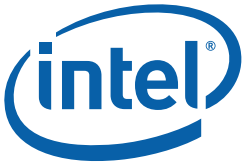
The following table lists which primitive topology types support the presence of Cut Indicies. When the Index Buffer has **Cut Index Enable** set, it is UNDEFINED to issue a 3DPRIMITIVE with a primitive topology type not supporting a Cut Index (even if no cut indicies are actually present in the index buffer).

Definition	Cut Index?
3DPRIM_POINTLIST	Y
3DPRIM_LINELIST	Y
3DPRIM_LINESTRIP	Y
3DPRIM_TRILIST	Y
3DPRIM_TRISTRIP	Y
3DPRIM_TRIFAN	N
3DPRIM_QUADLIST	N
3DPRIM_QUADSTRIP	N
3DPRIM_TRISTRIP_REVERSE	Y
3DPRIM_POLYGON	N
3DPRIM_RECTLIST	N
3DPRIM_LINELOOP	N
3DPRIM_POINTLIST_BF	Y
3DPRIM_LINESTRIP_CONT	Y
3DPRIM_LINESTRIP_BF	Y
3DPRIM_LINESTRIP_CONT_BF	Y
3DPRIM_TRIFAN_NOSTIPPLE	N

3.3.2 Index Buffer Access

The figure below illustrates how the Index Buffer is accessed.


PitchInBytes
(function of IBState.IndexFormat)
IBState.StartingBufferAddress
3DPRIM.StartingVertexLocation
x PitchInBytes
IBInstanceRestartAddress
(restart here each instance)
3DPRIM.VertexCountPerInstance
Index[v0]
Index[v1]
...
Index[vn-1]



This page intentionally left blank.

3.4 Vertex Buffers (VBs)

The 3DSTATE_VERTEX_BUFFERs command is used to define *Vertex Buffers* (VBs) used in subsequent 3DPRIMITIVE commands.

Most input vertex data is sourced from memory-resident VBs. A VB is a 1D arrays of structures, where the size of the structure as defined by the VB's **BufferPitch**. VBs are accessed either as *VERTEXDATA buffers*, as defined by the VB's **BufferAccessType**. The VB's access type will determine whether the VF-computed VertexIndex is used to access data in the VB.

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

3.4.1 3DSTATE_VERTEX_BUFFERS

This command is used to specify VB state used by the VF function. From 1 to 17 VBs can be specified, where the **VertexBufferID** field within the VERTEX_BUFFER_STATE structure(s) indicate the specific VB. If a VB definition is not included in this command, its associated state is left unchanged and available for use if previously defined.

NOTES:

- For any 3DSTATE_VERTEX_BUFFERS command, at least one VERTEX_BUFFER_STATE structure must be included.
- VERTEX_BUFFER_STATE structures are 4 DWords for VERTEXDATA.
- Inclusion of partial VERTEX_BUFFER_STATE structures is UNDEFINED.

The order in which VBs are defined within this command can be arbitrary, though a vertex buffer must be defined only once in any given command (otherwise operation is UNDEFINED).

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 03h
	28:16	**GFXPIPE Opcode =** 3DSTATE_VERTEX_BUFFERS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 08h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	**DWord Length** (excludes DWords 0,1) 4n-1 (where n = # of buffer states included)
1-4		**Vertex Buffer State [0]** Format: VERTEX_BUFFER_STATE
5-8		**Vertex Buffer State [1]**
…		…
(4n-3)-(4n)		**Vertex Buffer State [..]**

3.4.2 VERTEX_BUFFER_STATE Structure

This structure is used in 3DSTATE_VERTEX_BUFFERS to set the state associated with a VB. The VF function will use this state to determine how/where to extract vertex element data for all vertex elements associated with the VB.

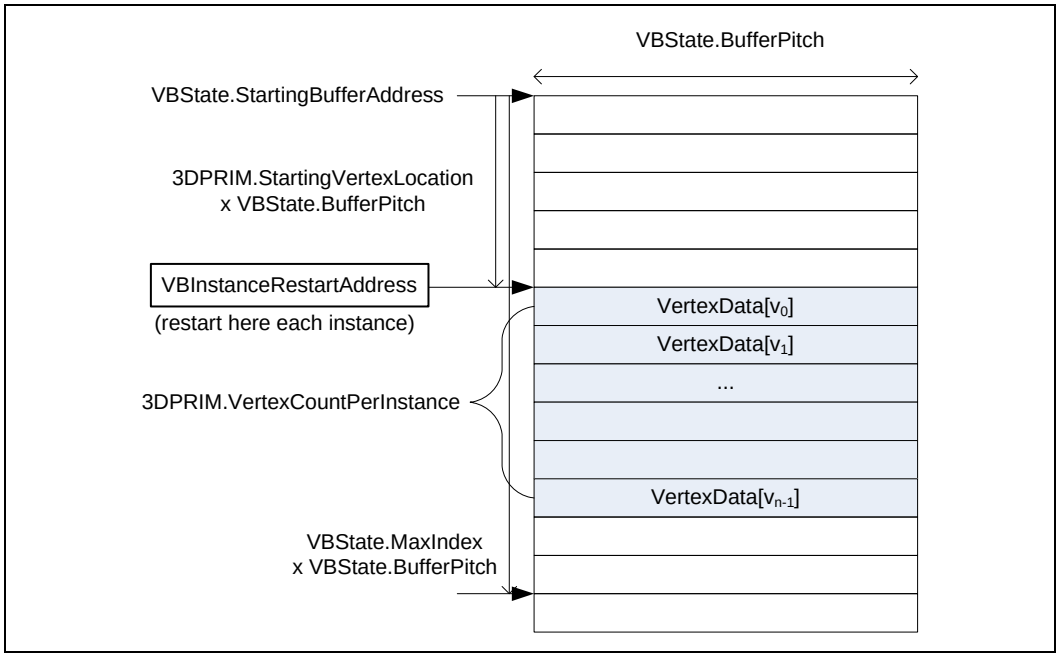
The VERTEX_BUFFER_STATE structure is 4 DWords for VERTEXDATA buffers.

A VB is defined as a 1D array of vertex data structures, accessed via a computed index value. The VF function therefore needs to know the starting address of the first structure (index 0) and size of the vertex data structure. If an index value outside of the range [0,Max Index] is used to access this vertex buffer, the value 0 is returned.

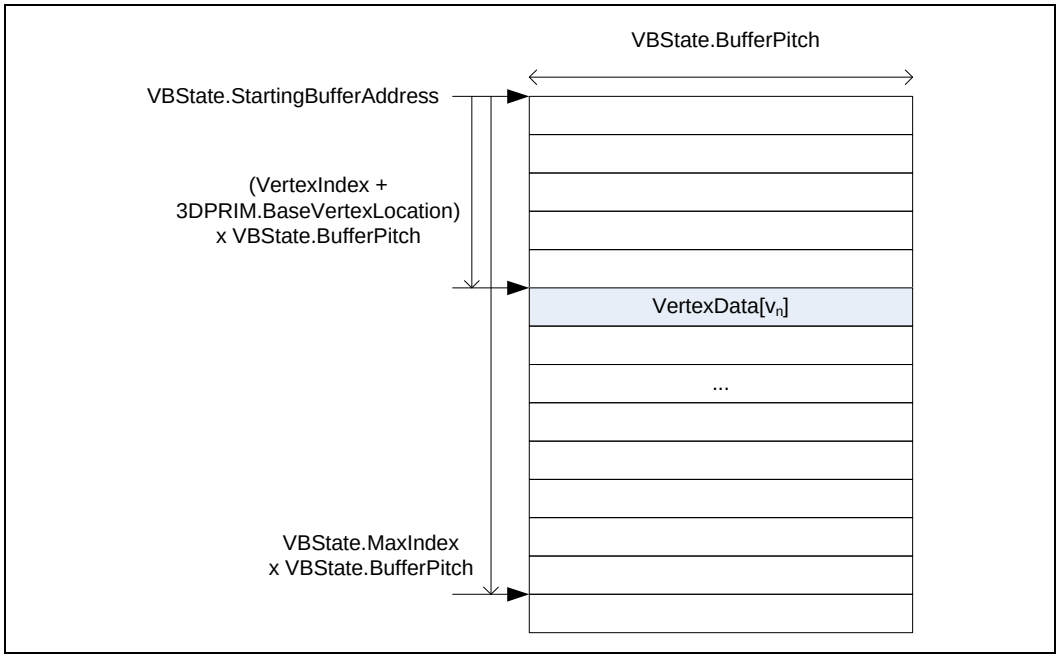
DWord	Bits(#)	Description
0	31:27	**Vertex Buffer Index:** This field contains an index value which selects the VB state being defined. Format: U5 index Range: [0,16]
	26	**Buffer Access Type:** This field determines how vertex element data is extracted from this VB. This control applies to all vertex elements associated with this VB. 0 = VERTEXDATA – For SEQUENTIAL vertex access, each vertex is sourced from sequential structures within the VB. For RANDOM vertex access, each vertex is looked up (separately) via a computed index value.
	25:11	**Reserved: MBZ**
	10:0	**Buffer Pitch:** This field specifies the pitch in bytes of the structures accessed within the VB. This information is required in order to access elements in the VB via a structure index. **Programming Notes:** • Different VERTEX_BUFFER_STATE structures can refer to the same memory region using different **Buffer Pitch** values. • See note on 64-bit float alignment in **Buffer Starting Address**. Format: U11 count of Bytes Range: [0,2047] Bytes

DWord	Bits(#)	Description
1	31:0	**Buffer Starting Address:** This field contains the byte-aligned Graphics Address of the first element of interest within the VB. Software must program this value with the combination (sum) of the base address of the memory resource and the byte offset from the base address to the starting structure within the buffer. **Programming Notes:** • 64-bit floating point values must be 64-bit aligned in memory, or UNPREDICTABLE data will be fetched. When accessing an element containing 64-bit floating point values, the **Buffer Starting Address** and **Source Element Offset** values must add to a 64-bit aligned address, and **BufferPitch** must be a multiple of 64-bits. • VBs can only be allocated in linear (not tiled) graphics memory. • VBs can only be mapped to Main Memory (UC). They must not be mapped to snooped System Memory, or UNPREDICTABLE values may be read. • As computed index values are, by definition, interpreted as unsigned values, there is no issue with accesses to locations before (lower address value) the start of the buffer. However, these wrapped indices are subject to Max Index checking (see below). Format: GraphicsAddress[31:0]
2	31:0	**Max Index:** This field defines the maximum (inclusive) structure index accessible for this particular VB. **Programming Notes:** • Use of an index larger than the Max Index returns 0. This includes a “negative” computed index which, when viewed as an unsigned value, exceeds Max Index. Format: U32
3	31:0	Reserved: MBZ

3.4.3 VERTEXDATA Buffers – SEQUENTIAL Access



3.4.4 VERTEXDATA Buffers – RANDOM Access



3.5 Input Vertex Definition

The 3DSTATE_VERTEX_ELEMENTS command is used to define the source and format of input vertex data and the format of how it is stored in the destination VUE as part of 3DPRIMITIVE processing in the VF unit.

Refer to *3DPRIMITIVE Processing* below for the general flow of how input vertices are input and stored during processing of the 3DPRIMITIVE command.

3.5.1 3DSTATE_VERTEX_ELEMENTS

This is a variable-length command used to specify the active vertex elements (up to 18). Each VERTEX_ELEMENT_STATE structure contains a **Valid** bit which determines which elements are used.

RESTRICTIONS/NOTES:

- At least one VERTEX_ELEMENT_STATE structure must be included.
- Vertex elements must be ordered by increasing **Destination Element Offset**.
- Inclusion of partial VERTEX_ELEMENT_STATE structures is UNDEFINED.
- SW must ensure that at least one vertex element is defined prior to issuing a 3DPRIMTIVE command, or operation is UNDEFINED.
- There are no 'holes' allowed in the destination vertex: NOSTORE components must be overwritten by subsequent components unless they are the trailing DWords of the vertex. Software must explicitly chose some value (probably 0) to be written into DWords that would otherwise be 'holes'.
- (See additional restrictions listed in the command fields and VERTEX_ELEMENT_STATE description).

DWord	Bit	Description
0	31:29	**Command Type** = GFXPIPE = 03h
	28:16	**GFXPIPE Opcode** = 3DSTATE_VERTEX_ELEMENTS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 09h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (excludes DWords 0,1) Vertex Element Count = (DWord Length + 1) / 2
1-2		Element[0] Format: VERTEX_ELEMENT_STATE
[3-4]		Element[1]
...		...
[35-36]		Element[17]

3.5.2 VERTEX_ELEMENT_STATE Structure

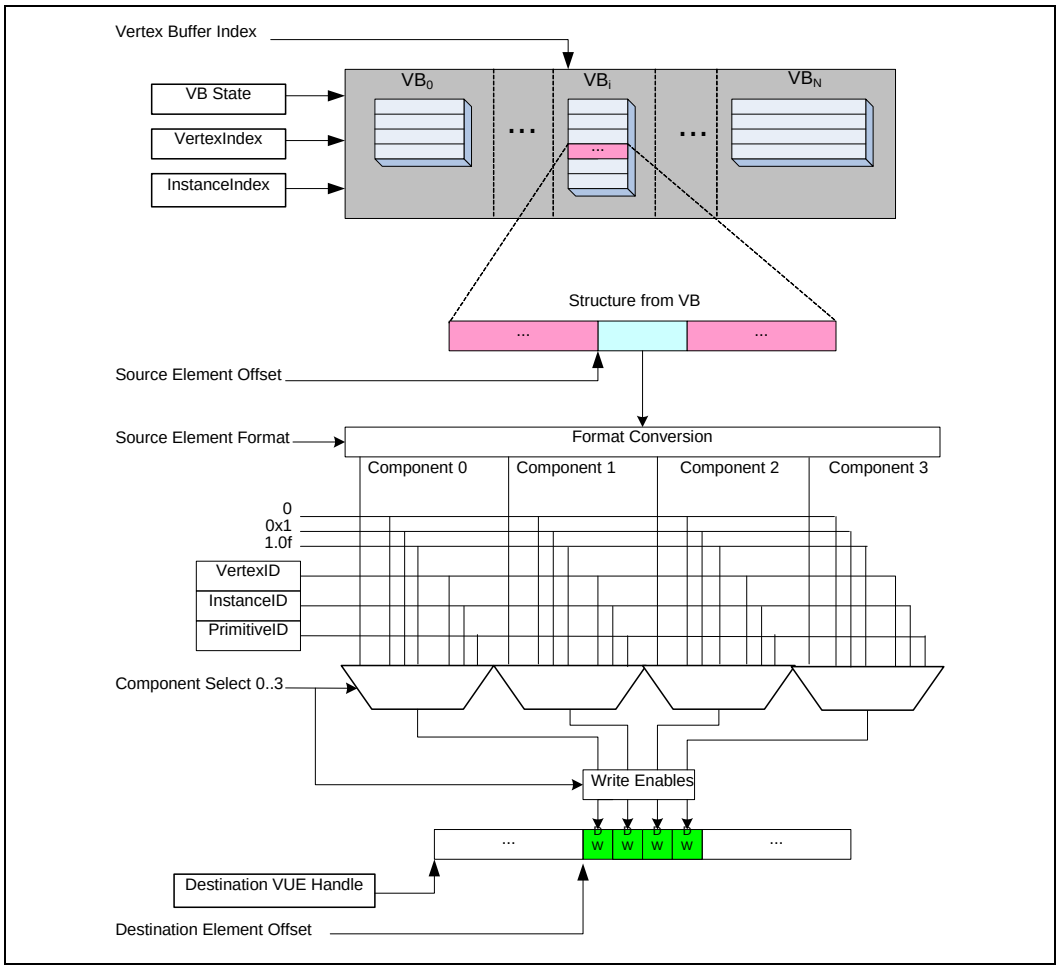
This structure is used in 3DSTATE_VERTEX_ELEMENTS to set the state associated with a *vertex element*. A vertex element is defined as an entity supplying from 1 to 4 DWord vertex components to be stored in the vertex URB entry. Up to 18 vertex elements are supported. The VF function will use this state, and possibly the state of the associated vertex buffer, to fetch/generate the source vertex element data, perform any required format conversions, padding with zeros, and store the resulting destination vertex element data into the vertex URB entry.

DWord	Bits(#)	Description
0	31:27	**Vertex Buffer Index:** This field specifies which vertex buffer the element is sourced from. Format: U5 Range: [0,16] (Up to 17 VBs are supported)
	26	**Valid:** If TRUE, this vertex element is used in vertex assembly If FALSE, this vertex element is not used. Format: Boolean
	25	Reserved: MBZ
	24:16	**Source Element Format:** This field specifies the format in which the memory-resident source data for this particular vertex element is stored in the memory buffer. This only applies to elements stored with VFCOMP_STORE_SRC component control. (All other component types have an explicit format). Format: The encoding of this field is identical the **Surface Format** field of the SURFACE_STATE structure, as described in the *Sampler* chapter. Range: Valid encodings are those marked as "Y" in the "Vertex Buffer" column of the table of Surface Format encodings in the *Sampler* chapter.
	15:12	Reserved: MBZ
	10:0	**Source Element Offset** (in bytes) Byte offset of the source vertex element data in the structures comprising the vertex buffer. **Programming Notes:** • See note on 64-bit float alignment in **Buffer Starting Address**. Format: U11 byte offset Range: [0,2047]
1	31	Reserved: MBZ

DWord	Bits(#)	Description
	30:28	**Component 0 Control:** This field specifies which value is stored for component 0 of this particular vertex element. 0: VFCOMP_NOSTORE: Don't store this component. (Not valid for Component 0, but can be used for Component 1-3). Once this setting is used for a component, all higher-numbered components (if any) MUST also use this setting. (I.e., no holes within any particular vertex element). Also, there are no 'holes' allowed in the destination vertex: NOSTORE components must be overwritten by subsequent components unless they are the trailing DWords of the vertex. Software must explicitly chose some value (probably 0) to be written into DWords that would otherwise be 'holes'. 1: VFCOMP_STORE_SRC: Store corresponding component from format-converted source element. Storing a component that is not included in the Source Element Format results in an UNPREDICTABLE value being stored. Software should used the STORE_0 or STORE_1 encoding to supply default components. 2: VFCOMP_STORE_0: Store 0 (interpreted as 0.0f if accessed as a float value) 3: VFCOMP_STORE_1_FP: Store 1.0f 4: VFCOMP_STORE_1_INT: Store 0x1 5: VFCOMP_STORE_VID: Store Vertex ID (as U32)
	27	Reserved: MBZ
	26:24	**Component 1 Control**
	23	Reserved: MBZ
	22:20	**Component 2 Control**
	19	Reserved: MBZ
	18:16	**Component 3 Control**
	15:8	Reserved: MBZ
	7:0	**Destination Element Offset:** This field specifies a DWord offset into the target URB Entry into which the converted vertex element is to be written. NOTES: • **Element Alignment**: Elements must be size-aligned within the URB entry. E.g., it is valid to pack two 2-component elements within one 128-bit region, but it is not valid to have a 4-component element to span two 128-bit regions. This restriction is imposed as a result of limitations on how logical elements can be accessed from GRF registers by the GEN4 EUs (it is not an restriction imposed by the VF unit). • The data written to the URB will be padded (with 0) to a 256-bit boundary.

3.5.3 Vertex Element Data Path

The following diagram shows the path by which a vertex element within the destination VUE is generated and how the fields of the VERTEX_ELEMENT_STATE structure are used to control the generation.



3.6 3D Primitive Processing

3.6.1 3DPRIMITIVE Command

The 3DPRIMITIVE command is used to submit 3D primitives to be processed by the 3D pipeline. Typically the processing results in the rendering of pixel data into the render targets, but this is not required.

The parameters passed in this command are forwarded to the Vertex Fetch function. The Vertex Fetch function will use this information to generate vertex data structures and store them in the URB. These vertices are then passed down the 3D pipeline for possible processing by the Vertex Shader, Geometry Shader, and Clipper. If rendering is required, the computed vertices are passed down to the StripFan and WindowerMasker units.

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**3D Command Opcode =** 3DPRIMITIVE 3D[28:27 = 3h; 26:24 = 3h; 23:16 = 00h]
	15	**Vertex Access Type:** This field specifies how data held in vertex buffers marked as VERTEXDATA is accessed by Vertex Fetch. Format = VertexAccessType = 0 = SEQUENTIAL VERTEXDATA buffers are accessed sequentially. 1 = RANDOM VERTEXDATA buffers are accessed randomly via an index obtained from the Index Buffer.
	14:10	**Primitive Topology Type:** This field specifies the *topology type* of 3D primitive generated by this command. Note that a single primitive topology (list/strip/fan/etc.) can contain a number of basic objects (lines, triangles, etc.). Format = 3D_PrimTopoType (see table below for encoding, see *3D Overview* for diagrams and general comments)
	9	Reserved: MBZ
	8	Reserved: MBZ
	7:0	**DWord Length** (excludes DWords 0,1) = 4
1	31:0	**Vertex Count:** This field specifies how many vertices are to be generated for the primitive topology. Programming Notes: • This value should specify a valid number of vertices for the primitive topology type. However, in cases where too few or too many vertices are provided, the unused vertices will be silently discarded by the pipeline. • A 0 value is this field effectively makes the command a 'no-operation'. If **Internal Vertex Count** is clear): Format = U32 count of vertices Range = [0, 2^32-1] (upper limit probably constrained by VB size)

DWord	Bit	Description
2	31:0	**Start Vertex Location:** This field specifies the “starting vertex”. This allows skipping over part of the vertices in a buffer if, for example, a previous 3DPRIMITIVE command had already drawn the primitives associated with the earlier entries. For SEQUENTIAL access, this field specifies a starting structure index into the vertex buffers For RANDOM access, this field specifies a starting index into the Index Buffer. **Programming Note:** • Access of any data outside of the valid extent of a vertex or index buffer will return the value 0 (i.e., appears as if the data stored at the invalid location was 0). Format = U32 structure index
3	31:0	Reserved: Must be set to 1
4	31:0	Reserved: MBZ
5	31:0	**Base Vertex Location:** This field specifies a signed bias to be added to values read from the index buffer. This allows the same index buffer values to access different vertex data for different commands. This field applies only to RANDOM access mode. This field is ignored for SEQUENTIAL access mode, where there Start Vertex Location can be used to specify different regions in the vertex buffers. **Programming Note:** • Access of any data outside of the valid extent of a vertex or index buffer will return the value 0 (i.e., appears as if the data stored at the invalid location was 0). Format = S31 structure index bias

The following table defines the encoding of the Primitive Topology Type field. See *3D Pipeline* for details, programming restrictions, diagrams and a discussion of the basic primitive types.

Table 3-1. 3D Primitive Topology Type Encoding

Encoding	Definition
00h	Reserved
01h	**3DPRIM_POINTLIST**
02h	**3DPRIM_LINELIST**
03h	**3DPRIM_LINESTRIP**
04h	**3DPRIM_TRILIST**
05h	**3DPRIM_TRISTRIP**
06h	**3DPRIM_TRIFAN**
07h	**3DPRIM_QUADLIST**
08h	**3DPRIM_QUADSTRIP**
09h-0Ch	Reserved
0Dh	**3DPRIM_TRISTRIP_REVERSE**
0Eh	**3DPRIM_POLYGON**
0Fh	**3DPRIM_RECTLIST**
10h	**3DPRIM_LINELOOP**
11h	**3DPRIM_POINTLIST_BF**
12h	**3DPRIM_LINESTRIP_CONT**
13h	**3DPRIM_LINESTRIP_BF**
14h	**3DPRIM_LINESTRIP_CONT_BF**
15h	Reserved
16h	**3DPRIM_TRIFAN_NOSTIPPLE**
17h-1Fh	Reserved

3.6.2 Functional Overview

The following pseudocode summarizes the general flow of 3D Primitive Processing.

```
VertexLoop {
   VertexIndexGeneration
      OutputBufferedVertex
      VertexCacheLookup
      if (miss) {
         VertexElementLoop {
            SourceElementFetch
            FormatConversion
            DestinationComponentSelection
            PrimitiveInfoGeneration
            URBWrite
         }
      }
}
TerminatePrimitive
```

3.6.3 VertexLoop

The VertexLoop iterates `VertexNumber` through the `VertexCount` vertices.

For each iteration, a number of processing steps are performed (see below) to generate the information that comprises a vertex. When a vertex is to be output, the following information is generated for that vertex:

- `PrimitiveType` associated with the vertex. This is simply a copy of the PrimitiveTopologyType field of the 3DPRIMITIVE
- VUE handle at which the vertex data is stored
  - For a Vertex Cache hit, the VUE handle is marked with a `VCHit` boolean, so that the VS unit will not attempt to process (shade) that vertex.
  - Otherwise, the VertexLoop will generate and store the input vertex data into the VUE referenced by this handle.
- `PrimStart` and `PrimEnd` booleans associated with the vertex. See PrimitiveInfoGeneration.

(Note that a single vertex of buffering is required in order to associate `PrimEnd` with a vertex, as this information may not be known until the next iteration through the VertexLoop (see *OutputPrimitiveDelimiter*).

`VertexNumber` value is incremented by 1 at the end of the loop.

3.6.4 VertexIndexGeneration

A `VertexIndex` value needs to be derived for each vertex. This index value is used as the vertex cache tag and will be used as a structure index into all VERTEXDATA VBs.

For SEQUENTIAL accessing, the `VertexIndex` value is derived as shown below:

```
VertexIndex = StartVertexLocation + VertexNumber
```

For RANDOM access, the VertexIndex is derived from an IBValue read from the IB, as shown below:

```
IBIndex = StartVertexLocation + VertexNumber
VertexIndex = IB[IBIndex] + BaseVertexLocation
```

3.6.5 VertexCacheLookup

The VertexIndex value is used as the tag value for the VertexCache (see *Vertex Cache*, above). If the Vertex Cache is enabled and the VertexIndex value hits in the cache, the VUE handle is read from the cache and inserted into the vertex stream. It is marked with a VCHit Boolean to surpress processing (shading) in the VS unit.

Otherwise, for Vertex Cache misses, a VUE handle is obtained to provide storage for the generated vertex data. VertexLoop processing then proceeds to iterate through the VEs to generate the destination VUE data.

3.6.6 VertexElementLoop

The VertexElementLoop generates and stores vertex data in the destination VUE one VE at a time.

Note that VEs must be defined (via 3DSTATE_VERTEX_ELEMENTS) in order of increasing **Destination Element Offset**, though architecturally the order by which VEs are processed is arbitrary (has no impact on the results).

3.6.7 SourceElementFetch

The following assumes the VE requires data from a VB, which is the typical case.

The structure index within the VE's selected VB is computed as follows:

```
VBIndex = VertexIndex
```

If VBIndex is invalid (i.e., negative or past **Max Index**), the data returned from the VB fetch is defined to be zero. Otherwise, the address of the source data required for the VE is then computed and the data is read from the VB. The amount of data read from the VB is determined by the **Source Element Format**.

```
if ( (VBIndex<0) || (VBIndex>VB.MaxIndex) )
      srcData = 0
else
      pSrcData = VB.BufferStartingAddress + (VBIndex *
VB.BufferPitch) + VE.SourceElementOffset
      srcData = MemoryRead( pSrcData, VE.SourceElementFormat )
endif
```

3.6.8 FormatConversion

Once the VE source data has been fetched, it is subjected to format conversion. The output of format conversion is up to 4 32-bit components, each either integer or floating-point (as specified by the **Source Element Format**). See *Sampler* for conversion algorithms.

The following table lists the valid **Source Element Format** selections, along with the format and availability of the converted components (if a component is listed as "-", it cannot be used as source of a VUE component). Note: This table is a subset of the list of supported surface formats defined in the *Sampler* chapter. Please refer to that table as the "master list". This table is here only to identify the components available (per format) and their format.

Table 3-2. Source Element Formats supported in VF Unit

Source Element Format	Converted Component				
	Format	0	1	2	3
256 bits					

Source Element Format	Converted Component				
	Format	0	1	2	3
R64G64B64A64_FLOAT	FLOAT	R	G	B	A
192 bits					
R64G64B64_FLOAT	FLOAT	R	G	B	A
128 bits					
R32G32B32A32_FLOAT	FLOAT	R	G	B	A
R32G32B32A32_SNORM	FLOAT	R	G	B	A
R32G32B32A32_UNORM	FLOAT	R	G	B	A
R32G32B32A32_SINT	SINT	R	G	B	A
R32G32B32A32_UINT	UINT	R	G	B	A
R32G32B32A32_SSCALED	FLOAT	R	G	B	A
R32G32B32A32_USCALED	FLOAT	R	G	B	A
R64G64_FLOAT	FLOAT	R	G	-	-
96 bits					
R32G32B32_FLOAT	FLOAT	R	G	B	-
R32G32B32_SNORM	FLOAT	R	G	B	-
R32G32B32_UNORM	FLOAT	R	G	B	-
R32G32B32_SINT	SINT	R	G	B	-
R32G32B32_UINT	UINT	R	G	B	-
R32G32B32_SSCALED	FLOAT	R	G	B	-
R32G32B32_USCALED	FLOAT	R	G	B	-
64 bits					
R16G16B16A16_FLOAT	FLOAT	R	G	B	A
R16G16B16A16_SNORM	FLOAT	R	G	B	A
R16G16B16A16_UNORM	FLOAT	R	G	B	A
R16G16B16A16_SINT	SINT	R	G	B	A
R16G16B16A16_UINT	UINT	R	G	B	A
R16G16B16A16_SSCALED	FLOAT	R	G	B	A
R16G16B16A16_USCALED	FLOAT	R	G	B	A
R32G32_FLOAT	FLOAT	R	G	-	-
R32G32_SNORM	FLOAT	R	G	-	-
R32G32_UNORM	FLOAT	R	G	-	-
R32G32_SINT	SINT	R	G	-	-
R32G32_UINT	UINT	R	G	-	-
R32G32_SSCALED	FLOAT	R	G	-	-
R32G32_USCALED	FLOAT	R	G	-	-
R64_FLOAT	FLOAT	R	-	-	-
48 bits					

Source Element Format	Converted Component				
	Format	0	1	2	3
R16G16B16_SNORM	FLOAT	R	G	B	-
R16G16B16_UNORM	FLOAT	R	G	B	-
R16G16B16_SSCALED	FLOAT	R	G	B	-
R16G16B16_USCALED	FLOAT	R	G	B	-
32 bits					
R10G10B10A2_UNORM	FLOAT	R	G	B	A
R10G10B10A2_UINT	UINT	R	G	B	A
R10G10B10X2_USCALED	FLOAT	R	G	B	-
R10G10B10_SNORM_A2_UNORM	FLOAT	R	G	B	A
B8G8R8A8_UNORM	FLOAT	B	G	R	A
R8G8B8A8_SNORM	FLOAT	R	G	B	A
R8G8B8A8_UNORM	FLOAT	R	G	B	A
R8G8B8A8_SINT	SINT	R	G	B	A
R8G8B8A8_UINT	UINT	R	G	B	A
R8G8B8A8_SSCALED	FLOAT	R	G	B	A
R8G8B8A8_USCALED	FLOAT	R	G	B	A
R11G11B10_FLOAT	FLOAT	R	G	B	-
R16G16_FLOAT	FLOAT	R	G	-	-
R16G16_SNORM	FLOAT	R	G	-	-
R16G16_UNORM	FLOAT	R	G	-	-
R16G16_SINT	SINT	R	G	-	-
R16G16_UINT	UINT	R	G	-	-
R16G16_SSCALED	FLOAT	R	G	-	-
R16G16_USCALED	FLOAT	R	G	-	-
R32_FLOAT	FLOAT	R	-	-	-
R32_SINT	SINT	R	-	-	-
R32_UINT	UINT	R	-	-	-
R32_SSCALED	FLOAT	R	-	-	-
R32_USCALED	FLOAT	R	-	-	-
R32_SNORM	FLOAT	R	-	-	-
R32_UNORM	FLOAT	R	-	-	-
24 bits					
R8G8B8_SNORM	FLOAT	R	G	B	-
R8G8B8_UNORM	FLOAT	R	G	B	-
R8G8B8_SSCALED	FLOAT	R	G	B	-
R8G8B8_USCALED	FLOAT	R	G	B	-
16 bits					

Source Element Format	Converted Component				
	Format	0	1	2	3
R8G8_SNORM	FLOAT	R	G	-	-
R8G8_UNORM	FLOAT	R	G	-	-
R8G8_SINT	SINT	R	G	-	-
R8G8_UINT	UINT	R	G	-	-
R8G8_SSCALED	FLOAT	R	G	-	-
R8G8_USCALED	FLOAT	R	G	-	-
R16_FLOAT	FLOAT	R	-	-	-
R16_SNORM	FLOAT	R	-	-	-
R16_UNORM	FLOAT	R	-	-	-
R16_SINT	SINT	R	-	-	-
R16_UINT	UINT	R	-	-	-
R16_SSCALED	FLOAT	R	-	-	-
R16_USCALED	FLOAT	R	-	-	-
8 bits					
R8_SNORM	FLOAT	R	-	-	-
R8_UNORM	FLOAT	R	-	-	-
R8_SINT	SINT	R	-	-	-
R8_UINT	UINT	R	-	-	-
R8_SSCALED	FLOAT	R	-	-	-
R8_USCALED	FLOAT	R	-	-	-

3.6.9 DestinationFormatSelection

The **Component Select 0..3** bits are then used to select, on a per-component basis, which destination components will be written and with which value. The supported selections are the converted source component, ,the constants 0 or 1.0f, or nothing (VFCOMP_NO_STORE). If a converted component is listed as '-' (not available) in Table 3-2, it must not be selected (via VFCOMP_STORE_SRC), or an UNPREDICTABLE value will be stored in the destination component.

The selection process sequences from component 0 to 3. Once a **Component Select** of VFCOMP_NO_STORE is encountered, all higher-numbered **Component Select** settings must also be programmed as VFCOMP_NO_STORE. It is therefore not permitted to have 'holes' in the destination VE.

3.6.10 URBWrite

The selected destination components are written into the destination VUE starting at **Destination Offset Select**. See the description of 3DPRIMITIVE for restrictions on this field.

3.6.11 OutputBufferedVertex

In order to accommodate ‘cut’ processing, the VF unit buffers one output vertex. The generation of a new vertex or the termination of a primitive causes the buffered vertex to be output to the pipeline.

3.7 Dangling Vertex Removal

The last functional stage of processing of the 3DPRIMITIVE command is the removal of “dangling” vertices. This includes the discarding of primitive topologies without enough vertices for a single object (e.g., a TRISTRIP with only two vertices), as well as the discarding of trailing vertices that do not form a complete primitive (e.g., the last two vertices of a 5-vertex TRILIST).

This function is best described as a filter operating on the vertex stream emitted from the processing of the 3DPRIMITIVE. The filter inputs the PrimType, PrimStart and PrimEnd values associated with the generated vertices. The filter only outputs primitive topologies without dangling vertices. This requires the filter to (a) be able to buffer some number of vertices, and (b) be able to remove dangling vertices from the pipeline and dereference the associated VUE handles.

§§

4 Vertex Shader (VS) Stage

4.1 VS Stage Overview

The VS stage of the GEN4 3D Pipeline is used to perform processing ("shading") of vertices after being assembled and written to the URB by the VF function. The primary function of the VS stage is to pass vertices that miss in the Vertex Cache to VS threads, and then pass the VS thread-generated vertices down the pipeline. Vertices that hit in the Vertex Cache are passed down the pipeline unmodified.

When the VS stage is disabled, vertices flow through the unit unmodified (i.e., as written by the VF unit).

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Overview* chapter for a general description of a 3D pipeline stage, as much of the VS stage operation and control falls under these "common" functions. I.e., most stage state variables and VS thread payload parameters are described in *3D Overview*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the VS stage, and any exceptions the VS stage exhibits with respect to common FF unit functions.

4.1.1 Vertex Caching

The 3D Pipeline employs a Vertex Cache that is shared between the VF and VS units. (See *Vertex Fetch* chapter for additional information). The Vertex Cache may be explicitly DISABLED via the **Vertex Cache Disable** bit in VS_STATE. Even when explicitly ENABLED, the VS unit can (by default) implicitly disable and invalidate the Vertex Cache when it detects the following condition:

1. Sequential indices are used in the 3DPRIMITIVE command (though this is effectively a don't care as there wouldn't be any hits anyway).

The implicit disable will persist as long as one of these conditions persist. The **Vertex Cache Implicit Disable Inhibit** bit in the VFSKPD MI register is provided to inhibit the VS unit's implicit cache disable. If inhibited, software is responsible for explicitly enabling/disabling the vertex cache as required for correct operation.

Note: Software can allow the implicit cache disable (the default action) and live with some possible performance penalty due to the too-often-disabled cache.

The following table summarizes the modes of operation of the Vertex Cache:

Vertex Cache	VS Function Enable	Mode of Operation
DISABLED (implicitly or explicitly)	DISABLED	Vertex Cache is not used. VF unit will assemble all vertices and write them into the URB entry supplied by the VS unit. VS unit will pass references to these VUEs down the pipeline unmodified.
DISABLED (implicitly or explicitly)	ENABLED	Vertex Cache is not used. VF unit will assemble all vertices and write them into the URB entry supplied by the VS unit. VS unit will spawn VS threads to process all vertices, overwriting the input data with the results. The VS unit pass references to these VUEs down the pipeline. Usage Model: This mode is only used when the VS function is required, but the VS kernel produces a side effect (e.g., writes to a memory buffer) which requires every vertex to be processed by a VS thread.
ENABLED	DISABLED	Vertex Cache is used to provide reuse of VF-generated vertices. The VF unit will check the cache and only process (assemble/write) vertices that miss in the cache. In either case, the VS unit will pass references to vertices (that hit or miss) down the pipeline without spawning any VS threads. Usage Model: Normal operation when the VS function is not required. Note that there may be situations which require the VS function to be used even when not explicitly required by the API. E.g., perspective divide may be required for clip testing.
ENABLED	ENABLED	Vertex Cache is used to provide reuse of VS-processed vertices. The VF unit will check the cache and only process (assemble/write) vertices that miss in the cache. The VS unit will only process (shade) the vertices that missed in the cache. The VS unit sends references to hit or missed vertices down the pipeline in the correct order. Usage Model: Normal operation when the VS function is required and use of the Vertex Cache is permissible.

4.2 VS Stage Input

As a stage of the GEN4 3D pipeline, the VS stage receives inputs from the previous (VF) stage. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the VS stage.

4.2.1 State

4.2.1.1 URB_FENCE

Refer to *3D Overview* for a description of how the VS stage processes this command.

4.2.1.2 VS_STATE

The following table describes the format and contents of the VS_STATE structure referenced by the **Pointer to VS State** field of the 3DSTATE_PIPELINED_POINTERS command.

DWord	Bit	Description
0	31:6	**Kernel Start Pointer:** This field specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address**. This field is ignored if **VS Function Enable** is DISABLED. **[DevBW-A,B] Errata BWT007:** Instructions pointed at by offsets from General State must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:6]
	5:4	Reserved: MBZ
	3:1	**GRF Register Count:** Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. This field is ignored if **VS Function Enable** is DISABLED. Format = U3 register block count - 1 Range = [0,7] = [16,128] GRF registers
	0	Reserved: MBZ

DWord	Bit	Description
1	31	**Single Program Flow (SPF) :** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). If set, the VS unit will only dispatch 1-vertex thread payloads. See CR0 description in *ISA Execution Environment*. This field is ignored if **VS Function Enable** is DISABLED. 0 = Multiple Program Flows (1- or 2-vertex threads spawned, operating under normal (SIMD4x2) mode) 1 = Single Program Flow (only 1-vertex threads spawned, operating under SPF EU mode)
	30:26	Reserved: MBZ
	25:18	**Binding Table Entry Count:** Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache. This field is ignored if **VS Function Enable** is DISABLED. Format = U8 Range = [0,255]
	17	**Thread Priority:** Specifies the priority of the thread for dispatch: This field is ignored if **VS Function Enable** is DISABLED. 0 = Normal Priority 1 = High Priority **Programming Notes:** • This field must be zero.
	16	**Floating Point Mode:** Specifies the initial floating point mode used by the dispatched thread. This field is ignored if **VS Function Enable** is DISABLED. 0 = Use IEEE-754 Rules 1 = Use alternate rules
	15:14	Reserved: MBZ
	13	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. This field is ignored if **VS Function Enable** is DISABLED. Format: Enable
	12	Reserved: MBZ
	11	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. This field is ignored if **VS Function Enable** is DISABLED. Format: Enable
	10:8	Reserved: MBZ

DWord	Bit	Description
	7	**Software Exceptio Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. This field is ignored if **VS Function Enable** is DISABLED. Format: Enable
	6:0	Reserved: MBZ
2	31:10	**Scratch Space Base Offset:** Specifies the starting location of the scratch space area allocated to this FF unit as a 1K-byte aligned offset from the **General State Base Address**. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space**. The computed offset of the thread-specific portion will be passed in the thread payload as **Scratch Space Offset**. The thread is expected to utilize "stateless" DataPort read/write requests to access scratch space, where the DataPort will cause the **General State Base Address** to be added to the offset passed in the request header. This field is ignored if **VS Function Enable** is DISABLED. Format = GeneralStateOffset[31:10]
	9:4	Reserved: MBZ
	3:0	**Per-Thread Scratch Space:** Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space. This field is ignored if **VS Function Enable** is DISABLED. Format = U4 power of 2 Bytes over 1K Bytes Range = [0,11] indicating [1K Bytes, 2M Bytes] **Programming Notes:** This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it.
3	31	Reserved : MBZ
	30:25	**Constant URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. This field is ignored if **VS Function Enable** is DISABLED. Format = U6 Range = [0,63]
	24	Reserved : MBZ
	23:18	**Constant URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload. This field is ignored if **VS Function Enable** is DISABLED. Format = U6 Range = [0,63]

DWord	Bit	Description
	17	Reserved : MBZ
	16:11	**Vertex URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. This field is ignored if **VS Function Enable** is DISABLED. **Programming Notes:** • It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread. Format = U6 Range = [1,63]
	10	Reserved : MBZ
	9:4	**Vertex URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread. This field is ignored if **VS Function Enable** is DISABLED. Format = U6 Range = [0,63]
	3:0	**Dispatch GRF Start Register for URB Data:** Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload. This field is ignored if **VS Function Enable** is DISABLED. Format = U4 Range = [0,15] indicating GRF [R0,R15]
4	31	Reserved : MBZ
	30:25	**Maximum Number of Threads:** Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. This field is ignored if **VS Function Enable** is DISABLED. Format = U5 representing thread count - 1 Range = [0,15] indicating thread count of [1,16]
	24	Reserved : MBZ
	23:19	**URB Entry Allocation Size:** Specifies the length of each URB entry owned by this FF unit. This field is always used (even if **VS Function Enable** is DISABLED). **Programming Note:** Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. Format = U5 count (of 512-bit units) – 1 Range = [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows
	18	Reserved: MBZ

DWord	Bit	Description
	17:11	**Number of URB Entries:** Specifies the number of URB entries that are used by this FF unit. This field is always used (even if **VS Function Enable** is DISABLED). **Programming Notes:** Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. Format = U6, see valid settings below Range = [8,12, 16, 32] (see restriction above) **DevBW-A,B Restriction**: Format = U6, see valid settings below Range = [8,12, 16]
	10	**Statistics Enable:** If ENABLED, this FF unit will engage in statistics gathering. See the ***Error! Reference source not found.*** section later in this chapter. If DISABLED, statistics information associated with this FF stage will be left unchanged. This field is effectively if **VS Function Enable** is DISABLED. Format: Enable
	9:0	Reserved : MBZ
5	31:5	**Sampler State Offset:** This field, together with the **General State Base Address**, specifies the starting location of the Sampler State Table used by threads spawned by this FF unit. It is specified as a 32-byte-granular offset from the **General State Base Address**. The Sampler will apply the offset to the **General State Base Address** when accessing Sampler State data. This field is ignored if **VS Function Enable** is DISABLED. **[DevBW-A,B] Errata BWT007:** Sampler state pointed at by offsets from General State must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5]
	4:3	Reserved : MBZ
	2:0	**Sampler Count:** Specifies how many samplers (in multiples of 4) the vertex shader 0 kernel uses. Used only for prefetching the associated sampler state entries. This field is ignored if **VS Function Enable** is DISABLED. Format = U3 Range = [0,4] 0 = no samplers used 1 = between 1 and 4 samplers used 2 = between 5 and 8 samplers used 3 = between 9 and 12 samplers used 4 = between 13 and 16 samplers used
6	31:2	Reserved : MBZ

DWord	Bit	Description
	1	**Vertex Cache Disable:** This bit controls the operation of the Vertex Cache. This field is always used. If the Vertex Cache is DISABLED and the VS Function is ENABLED, the Vertex Cache is not used and all incoming vertices will be passed to VS threads. If the Vertex Cache is ENABLED and the VS Function is ENABLED, incoming vertices that do not hit in the Vertex Cache will be passed to VS threads. If the Vertex Cache is ENABLED and the VS Function is DISABLED, input vertices that miss in the Vertex Cache will be assembled and written to the URB, though pass thru the VS stage unmodified (not shaded). The Vertex Cache is invalidated whenever the Vertex Cache becomes DISABLED , whenever the VS Function Enable toggles, between 3DPRIMITIVE commands. **Programming Notes:** • See the Vertex Caching section (above) for details on implicit vertex cache disabling and the enable/disable bit available to turn of any implicit disable. Format: Disable
	0	**VS Function Enable** If ENABLED, VS threads may be spawned to process VF-generated vertices before the resulting vertices are passed down the pipeline. If DISABLED, VF-generated vertices will pass thru the VS function and sent down the pipeline unmodified. The Vertex Cache is still available in this mode, if enabled. This field is always used. Format: Enable

4.2.2 Input Vertices

Refer to *3D Overview* for a description of the vertex information input to the VS stage.

4.3 VS Thread Request Generation

The following discussion assumes the VS Function is ENABLED.

When the Vertex Cache is disabled, the VS unit will pass each pair of incoming vertices to a VS thread. Under certain circumstances (e.g., prior to a state change or pipeline flush) the VS unit will spawn a VS thread to process a single vertex. Note that, in this case, the “unused” vertex slot will be “disabled” via the Execution Mask provided by the VS unit to the GEN4 subsystem as part of the thread dispatch (See ISA doc). The VS thread will in itself be unaware of the single-vertex case, and therefore a single VS kernel can be used to process one or two vertices. (The performance of single-vertex processing will roughly equal the two-vertex case).

When the Vertex Cache is enabled, the VF unit will detect vertices that hit in the cache and mark these vertices so that they will bypass VS thread processing and be output via a reference to the cached VUE. The VS unit will keep track of these cache-hit vertices as it proceeds to process cache-miss vertices. The VS unit guarantees that vertices will exit the unit in the order they are received. This may require the VS unit

to issue single-vertex VS threads to process a cache-miss vertex that has yet to be paired up with another cache-miss vertex (if this condition is preventing the VS unit from producing any output).

4.3.1 Thread Payload

The following table describes the payload delivered to VS threads.

Table 4-1. VS Thread Payload

DWord	Bit	Description
R0.7	31	**Snapshot Flag** If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID:** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. (See *Debug* for further description). Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Offset:** Specifies the of the scratch space allocated to the thread, specified as a 1KB-granular offset from the **General State Base Address**. See **Scratch Space Base Offset** description in VS_STATE. (See *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9:4	Reserved
	3:0	**FFTID:** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = GeneralStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). (See *3D Pipeline* for further description). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:0	Reserved : delivered as zeros (reserved for message header fields)
R0.1	31:25	Reserved

DWord	Bit	Description
	24:16	**Handle ID 1:** This ID is assigned by the FF unit and used to identify the URB Return Handle 1 to the FF unit (as FF-specific index value, not a URB address). If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). (See *Generic FF Unit* for further description). Format = Reserved for HW Implementation Use.
	15:9	Reserved
	8:0	**URB Return Handle 1:** This is the URB handle where the EU's upper channels (DWords 7:4) results are to be stored. If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). (See *Generic FF Unit* for further description). Format: U9 opaque handle
R0.0	31:25	Reserved
	24:16	**Handle ID 0:** This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). (See *Generic FF Unit* for further description). Format = Reserved for HW Implementation Use.
	15:9	Reserved
	8:0	**URB Return Handle 0:** This is the URB handle where the EU's lower channels (DWords 3:0) results are to be stored. (See *Generic FF Unit* for further description). Format: U9 opaque handle
[Varies] optional	31:0	**Constant Data (optional) :** Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state. The Constant Data arrives in a non-interleaved format.
Varies	31:0	**Vertex Data :** Data from (possibly) one or (more typically) two Vertex URB Entries is passed to the thread in the thread payload. The **Vertex URB Entry Read Offset** and **Vertex URB Entry Read Length** state variables define the regions of the URB entries that are read from the URB and passed in the thread payload. These SVs can be used to provide a subset of the URB data as required by SW. The vertex data is laid out in the thread header in an interleaved format. The lower DWords (0-3) of these GRF registers always contain data from a Vertex URB Entry. The upper DWords (4-7) may contain data from another Vertex URB Entry. This allows two vertices to be processed (shaded) in parallel SIMD8 fashion. The VS kernel is not aware of the validity of the upper vertex.

4.4 VS Thread Execution

A VS kernel (with one exception mentioned below) assumes it is to operate on two vertices in parallel. Input data is either passed directly in the thread payload (including the input vertex data) or indirectly via pointers passed in the payload.

Refer to *ISA* chapters for specifics on writing kernels that operate in SIMD4x2 fashion.

Refer to 3D Pipeline Stage Overview (*3D Overview*) for information on FF-unit/Thread interactions.

In the (unlikely) event that the VS kernel needs to determine whether it is processing one or two vertices, the kernel can compare the **URB Return Handle 0** and **URB Return Handle 1** fields of the thread payload. These fields will be different if two vertices are being processed and identical if one vertex is being processed. An example of when this test may be required is if the kernel outputs some vertex-dependent results into a memory buffer – without the test the single vertex case might incorrectly output two sets of results. Note that this is not the case for writing the URB destinations, as the Execution Mask will prevent the write of an undefined output.

4.4.1 Vertex Output

VS threads must always write the destination URB handles passed in the payload. VS threads are not permitted to request additional destination handles. Refer to 3D Pipeline Stage Overview (*3D Overview*) for details on how destination vertices are written and any required contents/formats.

4.4.2 Thread Termination

VS threads must signal thread termination, in all likelihood on the last message output to the URB shared function. Refer to the *ISA* doc for details on End-Of-Thread indication.

4.5 Primitive Output

The VS unit will produce an output vertex reference for every input vertex reference received from the VF unit, in the order received. The VS unit simply copies the `PrimitiveType`, `StartPrim`, and `EndPrim` information associated with input vertices to the output vertices, and does not use this information in any way. Neither does the VS unit perform any readback of URB data.

§§

5 Geometry Shader (GS) Stage

5.1 GS Stage Overview

The GS stage of the GEN4 3D Pipeline is used to convert objects within incoming primitives into new primitives through use of a spawned GEN4 thread. When enabled, the GS unit buffers incoming vertices, assembles the vertices of each individual object within the primitives, and passes these object vertices (along with other data) to the GEN4 subsystem for processing by a GS thread.

When the GS stage is disabled, vertices flow through the unit unmodified, with the exception that the Vertex Header of each vertex is read back from the URB and passed along with the vertex to the next (CLIP) stage.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Pipeline* chapter for a general description of a 3D Pipeline stage, as much of the GS stage operation and control falls under these “common” functions. I.e., most stage state variables and GS thread payload parameters are described in *3D Pipeline*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the GS stage, and any exceptions the GS stage exhibits with respect to common FF unit functions.

5.2 GS Stage Input

As a stage of the GEN4 3D pipeline, the GS stage receives inputs from the previous (VS) stage. Refer to *3D Pipeline* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the GS stage.

5.2.1 State

5.2.1.1 GS_STATE

The following table describes the format and contents of the GS_STATE structure referenced by the **Pointer to GS State** field of the 3DSTATE_PIPELINED_POINTERS command.

GS_STATE		
Project:	All	
Controls the GS stage hardware.		
DWord	**Bit**	**Description**
0	31:6	**Kernel Start Pointer** Project: All Format: GeneralStateOffset[31:6] This field specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address** **[DevBW-A,B] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)
	5:4	**Reserved** Project: All Format: MBZ
	3:1	**GRF Register Count** Project: All Format: U3 register block count - 1 Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16.
	0	**Reserved** Project: All Format: MBZ
1	31	**Single Program Flow (SPF)** Project: All Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). **Value** **Name** **Description** **Project** 0h Reserved All 1h Enable Single Program Flow enabled All
	30:26	**Reserved** Project: All Format: MBZ

GS_STATE		
	25:18	**Binding Table Entry Count** Project: All Format: U8 Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache.
	17	Reserved. MBZ
	16	**Floating Point Mode** Project: All Specifies the initial floating point mode used by the dispatched thread. **Value** **Name** **Description** **Project** 0h Use IEEE-754 Rules All 1h Use alternate rules All
	15:14	**Reserved** Project: All Format: MBZ
	13	**Illegal Opcode Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	12	**Reserved** Project: All Format: MBZ
	11	**Mask Stack Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*.
	10:8	**Reserved** Project: All Format: MBZ
	7	**Software Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	6:0	**Reserved** Project: All Format: MBZ
2	31:10	**Scratch Space Base Pointer** Project: All Format: GeneralStateOffset[31:10] Specifies the location of the scratch space area allocated to this FF unit, specified as a 1KB-granular offset from the **General State Base Address**. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space**.
	9:4	**Reserved** Project: All Format: MBZ

GS_STATE

	3:0	**Per-Thread Scratch Space** Project: All Format: U4 power of 2 Bytes over 1K Bytes FormatDesc Range [0,11] indicating [1K Bytes, 2M Bytes] Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.
3	31	**Reserved** Project: All Format: MBZ
	30:25	**Constant URB Entry Read Length** Project: All Format: U6 FormatDesc Range [0,63] Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments.
	24	**Reserved** Project: All Format: MBZ
	23:18	**Constant URB Entry Read Offset** Project: All Format: U6 **FormatDesc** Range [0,63] Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload.
	17	**Reserved** Project: All Format: MBZ
	16:11	**Vertex URB Entry Read Length** Project: All Format: U6 FormatDesc Range [1,63] Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.
	10	**Reserved** Project: All Format: MBZ

GS_STATE		
	9:4	**Vertex URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread.
	3:0	**Dispatch GRF Start Register for URB Data** Project: All Format: U4 FormatDesc Range [0,15] indicating GRF [R0,R15] Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload.
4	31:30	**Reserved** Project: All Format: MBZ
	29:25	**Maximum Number of Threads** Project: All Format: U5 thread count – 1 Range Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. **Programming Notes:** • When running in dual-thread mode, the **Number of URB Entries** field must contain an even number. Each thread will be allocated one half the total number of entries. • A URB_FENCE command must be issued subsequent to any change to the value in this field (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See *Graphics Processing Engine* (Command Ordering Rules) Format = U5 representing thread count – 1 Range = [0,1] indicating thread count of [1,2]
	24	**Reserved** Project: All Format: MBZ

GS_STATE

	23:19	**URB Entry Allocation Size** Project: All Format: U5 count (of 512-bit units) – 1 Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows Specifies the length of each URB entry owned by this FF unit. **Programming Notes** — **Project** Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All
	18	**Reserved** Project: All Format: MBZ
	17:11	**Number of URB Entries** Project: All Format: U7 Count of URB entries Range [1,32] if GS enabled, otherwise ignored. Specifies the number of URB entries that are used by this FF unit. **Programming Notes** — **Project** When running in dual-thread mode, the **Number of URB Entries** field must contain an even number. Each thread will be allocated one half the total number of entries. — All If ENABLED, the GS stage must be allocated at least one URB entry — All Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All
	9:10	**Reserved** Project: All Format: MBZ
	8	**Reserved** Project: All Format: MBZ
	7:0	**Reserved** Project: All Format: MBZ
5	31:5	**Sampler State Pointer** Project: All Format: GeneralStateOffset[31:5] This field specifies the starting location of the Sampler State Table used by threads spawned by this FF unit. It is specified as a 32-byte-granular offset from the **General State Pointer.** **Errata** — **Description** — **Project** Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) — BW-A,B
	4:3	**Reserved** Project: All Format: MBZ

GS_STATE

	2:0	**Sampler Count** Project: All Format: U3 Specifies how many samplers (in multiples of 4) the geometry shader kernel uses. Used only for prefetching the associated sampler state entries. **Value** **Name** **Description** **Project** 0h — no samplers used — All 1h — between 1 and 4 samplers used — All 2h — between 5 and 8 samplers used — All 3h — between 9 and 12 samplers used — All 4h — between 13 and 16 samplers used — All 5h-7h — Reserved — All
6	31	**Reserved** Project: All Format: Enable
	30	**Reorder Enable** Project: All Format: Enable This bit controls whether the GS unit reorders TRISTRIP/TRISTRIP_REV vertices passed in the GS thread payload. If ENABLED, the GS unit will reorder the vertices for “odd-numbered” triangles originating from TRISTRIP topologies and “even-numbered” triangles originating from TRISTRIP_REV topologies. (Note that the first triangle is considered “triangle 0”, which is even-numbered). With respect to the PrimType passed in the GS thread payload, the GS unit passes TRISTRIP when the vertices are not reordered, and TRISTRIP_REV when the vertices are reordered (regardless of whether a TRISTRIP or TRISTRIP_REV topology was being processed) If DISABLED, TRISTRIP/TRISTRIP_REV vertices are not reordered, and always passed in the order they are received from the pipeline. The GS unit will still toggle PrimType on alternating (as described above) so that the GS thread can perform the reordering internally (or do whatever is necessary to account for the non-reordering of its input).
	29	**Reserved** Project: All Format: MBZ
	28	Reserved
	27:4	**Reserved** Project: All Format: MBZ
	3:0	**Maximum VPIndex** Project: All Format: U4 index value (# of viewports -1) This field specifies the maximum valid VPIndex value, corresponding to the number of active viewports. If the source of the VPIndex exceeds this maximum value, a VPIndex value of 0 is passed down the pipeline. Note that this clamping does not affect a VPIndex value stored in the URB.

5.3 Object Staging

The GS unit's Object Staging Buffer (OSB) accepts primitive topologies as a stream of incoming vertices, and spawns a thread for each individual object within the topology.

5.4 GS Thread Request Generation

5.4.1 Object Vertex Ordering

The following table defines the number and order of object vertices passed in the Vertex Data portion of the GS thread payload, assuming an input topology with *N* vertices. The ObjectType passed to the thread is, by default, the incoming PrimTopologyType. Exceptions to this rule (for the TRISTRIP variants) are called out.

5.4.2 GS Thread Payload

Table 5-1 shows the layout of the payload delivered to GS threads.

Refer to 3D Pipeline Stage Overview (*3D Pipeline*) for details on those fields that are common amongst the various pipeline stages.

Table 5-1. GS Thread Payload

GRF DWord	Bit	Description
R0.7	31	**Snapshot Flag.** If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. (See *Debug* for further description). Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:1	Reserved
	0	**FFTID.** This ID is assigned by the fixed function unit and is relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use

GRF DWord	Bit	Description
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = GeneralStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). (See *Generic Pipeline Stage* for further description). **Programming Notes:** • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:10	Reserved : delivered as zeros (reserved for message header fields)
	9	**Edge Indicator [1].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex2 to Vertex0 is an exterior edge of the polygon (i.e., this is the last or only triangle of the polygon). If clear, that edge is an interior edge. The kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved
	8	**Edge Indicator [0].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex0 to Vertex1 is an exterior edge of the polygon (i.e., this is the first or only triangle of the polygon). If clear, that edge is an interior edge. The kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved
	7	Reserved: MBZ
	6:5	Reserved
	4:0	**Primitive Topology Type.** This field identifies the Primitive Topology Type associated with the primitive containing this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the GS unit may toggle this value between TRISTRIP and TRISTRIP_REV, as described in 5.4.1. Format: See *3D Pipeline*
R0.1	31:0	Reserved
R0.0	31:23	Reserved

GRF DWord	Bit	Description
	22:16	**Handle ID.** This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. Format: Reserved for Implementation Use
	15:9	Reserved
	8:0	**URB Return Handle.** This is the initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the first destination URB entry.
[Varies] optional	31:0	**Constant Data (optional) :** Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state. The Constant Data arrives in a non-interleaved format.
Varies	31:0	**Vertex Data.** There can be up to 6 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state Vertex 0 DWord 0 is located at Rn.0, Vertex 0 DWord 1 is located at Rn.1, etc. Vertex 1 DWord 0 immediately follows the last DWord of Vertex 0, and so on.

5.5 GS Thread Execution

A GS thread is capable of performing arbitrary algorithms given the thread payload (especially vertex) data and associated data structures (binding tables, sampler state, etc.) as input. Output can take the form of vertices output to the FF pipeline (at the GS unit) and/or data written to memory buffers via the DataPort.

The primary usage models for GS threads include (possible combinations of):

- Compiled application-provided "GS shader" programs, specifying an algorithm to convert the vertices of an input object into some output primitives. For example, a GS shader may convert lines of a line strip into polygons representing a corresponding segment of a blade of grass centered on the line. Or it could output absolutely nothing, effectively terminating the pipeline at the GS stage.
- Driver-generated instructions used to write pre-clipped vertices into memory buffers (see Stream Output below). This may be required whether or not an app-provided GS shader is enabled.
- Driver-generated instructions used to emulate API functions not supported by specialized hardware. These functions might include (but are not limited to):
  - Conversion of API-defined topologies into topologies that can be rendered (e.g., LINELOOP→LINESTRIP, POLYGON→TRIFAN, QUADs→TRIFAN, etc.)
  - Emulation of "Polygon Fill Mode", where incoming polygons can be converted to points, lines (wireframe), or solid objects.
  - Emulation of wide/sprite points.

5.5.1 Vertex Output

The GS kernel will typically use the URB_WRITE message to output vertices and request additional handles. (Refer to the *3D Pipeline* chapter for a general discussion of how FF units output vertices, and the *URB* chapter for details on the use of the URB_WRITE message.)

The following table lists which primitive topology types are valid for output by a GS thread.

PrimTopologyType	Supported for GS Thread Output?
LINELIST	Yes
LINESTRIP	Yes
LINESTRIP_BF	Yes
LINESTRIP_CONT	Yes
LINESTRIP_CONT_BF	Yes
LINELOOP	No
POINTLIST	Yes
POINTLIST_BF	Yes

PrimTopologyType	Supported for GS Thread Output?
POLYGON	Yes
QUADLIST	No
QUADSTRIP	No
RECTLIST	Yes
TRIFAN	Yes
TRIFAN_NOSTIPPLE	Yes
TRILIST	Yes
TRISTRIP	Yes
TRISTRIP_REV	Yes

The GS thread is responsible for providing correct PrimType, PrimStart and PrimEnd information for each vertex output, in the same fashion as the Vertex Fetch unit. Given that the GS thread is likely performing an algorithm as specified by an application "geometry shader" program, where the algorithm dictates when and if a vertex is to be output, the GS thread is allowed to output incomplete primitives (too few or too many vertices). The downstream FF units will correctly handle any dangling vertices.

However, the PrimStart and PrimEnd indicators must be correct for all vertices, e.g., the last vertex of a topology must have PrimEnd set. This may require the GS thread to postpone completion of a vertex output operation until either the next vertex is encountered or the algorithm (not the thread) completes.

Take for example a GS shader that outputs tristrips and uses a "cut" instruction in some conditionally-executed code. When outputting a vertex, the thread can't predict whether or not a subsequent "cut" will cause the vertex to be the last one of the tristrip topology. And so, the PrimEnd status of that vertex can't be ascertained until a subsequent "cut" or "emit" or algorithm termination is encountered. On the other hand, depending on the robustness of the compiler, the output from simple shaders (no looping, or conditional operations relating to vertex output) could permit a priori knowledge of the PrimStart/PrimEnd values. (For example, a simple GS shader that unconditionally converts an input point to a 2-triangle tristrip.)

Note that, through use (clearing) of the **Complete** bit in the URB_WRITE message, is it possible to write a vertex to the URB yet delay the "complete" indication until later. The PrimType, PrimStart, and PrimEnd indications are not sampled by the FF pipeline until **Complete** is set. This relieves the GS thread from actually having to buffer the pending vertex.

A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a "retained" handle refers to a URB handle that (a) has been pre-allocated or allocated and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

5.5.2 Thread Termination

GS threads must terminate by sending a URB_WRITE message with the **EOT** and **Complete** bits set. The Used bit can be set (if outputting a VUE) or clear (if freeing a used VUE).

5.6 Vertex Header Readback

The GS unit performs a readback of the Vertex Header of each vertex exiting the GS stage (either passed through or generated by a GS thread) as this information is required by the next FF stage (CLIP). Software is responsible for ensuring that any required Vertex Header fields are valid at this point in the pipeline. See *Vertex Data Overview* for a description of the Vertex Header fields and how they are read-back and used by the GS unit.

5.7 Primitive Output

This section refers to output from the GS unit to the pipeline, not output from the GS thread.

The GS unit will output primitives (either passed-through or generated by a GS thread) in the proper order. This includes the buffering of a concurrent GS thread's output until the preceding GS thread terminates. Note that the requirement to buffer subsequent GS thread output until the preceding GS thread terminates has ramifications on determining the number of VUEs allocated to the GS unit and the number of concurrent GS threads allowed.

6 Clip Stage

6.1 CLIP Stage Overview

The CLIP stage of the GEN4 3D Pipeline is similar to the GS stage in that it can be used to perform general processing on incoming 3D objects via spawned GEN4 threads. However, the CLIP stage also includes specialized logic to perform a *ClipTest* function on incoming objects. These two usage models of the CLIP stage are outlined below.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Overview* chapter for a general description of a 3D Pipeline stage, as much of the CLIP stage operation and control falls under these "common" functions. I.e., many of the CLIP stage state variables and CLIP thread payload parameters are described in *3D Overview*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the CLIP stage, details on the ClipTest function, and any exceptions the CLIP stage exhibits with respect to common FF unit functions.

6.1.1 Clip Stage – General-Purpose Processing

Numerous state variable controls are provided to tailor the ClipTest function as required by the API or primitive characteristics. These controls allow a mode where all objects are passed to CLIP threads, and in this regard the CLIP stage can be used as a second GS stage. However, unlike the GS stage, primitives output by CLIP threads will not be subject to 3D Clipping, and therefore any clip-testing/clipping of these primitives (if required) would need to be performed by the CLIP thread itself.

6.1.2 Clip Stage – 3D Clipping

The ClipTest fixed function is provided to optimize the CLIP stage for support of generalized *3D Clipping*. The CLIP FF unit examines the position of incoming vertices, performs a fixed function *VertexClipTest* on these positions, and then examines the results for the vertices of each independent object in *ClipDetermination*.

The results of ClipDetermination indicate whether an object is to be processed by a thread (MustClip), discarded (TrivialReject) or passed down the pipeline unmodified (TrivialAccept). In the MustClip case, the spawned thread is responsible for performing the actual 3D Clipping algorithm. The CLIP thread is passed the source object vertex data and is able to output a new, arbitrary 3D primitive (e.g., the clipped primitive), or no output at all. Note that the output primitive is independent in that it is comprised of newly-generated VUEs, and does not share vertices with the source primitive or other CLIP-generated primitives.

New vertices produced by the CLIP threads are stored in the URB. Their Vertex Headers are then read from the VUEs in order to insert the relevant information into the 3D pipeline. The CLIP unit maintains the proper ordering of CLIP-generated primitives and any surrounding trivially-accepted primitives. The CLIP unit also supports multiple concurrent CLIP threads and maintains the proper ordering of the thread outputs as dictated by the order of the source objects.

The outgoing primitive stream is sent down the pipeline to the Strip/Fan (SF) FF stage (now including the read-back VUE Vertex Header data such as Vertex Rosition (NDC or screen space), RTAIndex, VPIndex, PointWidth) and control information (PrimType, PrimStart, PrimEnd) while the remainder of the vertex data remains in the VUE in the URB.

6.2 Concepts

This section provides an overview of 3D clip-testing and clipping concepts. It is provided as background material: some of the concepts impact HW functionality while others impact CLIP kernel functionality.

6.2.1 The Clip Volume

3D objects are optionally clipped to the *clip volume*. The clip volume is defined as the intersection of a set of *clip half-spaces*. Six of these half-spaces define the view volume, while additional, user-defined half-spaces can be employed to perform clipping (or at least culling) within the view volume.

The CLIP stage design will permit the enable/disable of certain subsets of these clip half-spaces. This capability can be used, for example, to disable viewport, guardband, and near and far clipping as required by the API and other conditions.

6.2.1.1 View Volume

The intersection of the six view half-spaces defines the *view volume*. The view volume is defined in 4D clip space coordinates as:

View Clip Plane	'Outside' Condition	
	4D Clip Space	NDC space, positive w
XMIN (NDC Left)	clip.x < -clip.w	ndc.x < -1
XMAX (NDC Right)	clip.w < clip.x	ndc.x > 1
YMIN (NDC Bottom)	clip.y < -clip.w	ndc.y < -1
YMAX (NDC top)	clip.w < clip.y	ndc.y > 1
ZMIN (NDC Near)	clip.z < -clip.w	ndc.z < -1.0
ZMAX (NDC Far)	clip.w < clip.z	ndc.z > 1.0

Note that, since the 2D (X,Y) extent of the projected view volume is subsequently mapped to the 2D pixel space viewport, the terms "viewport" and "view volume" are used somewhat interchangeably in this discussion.

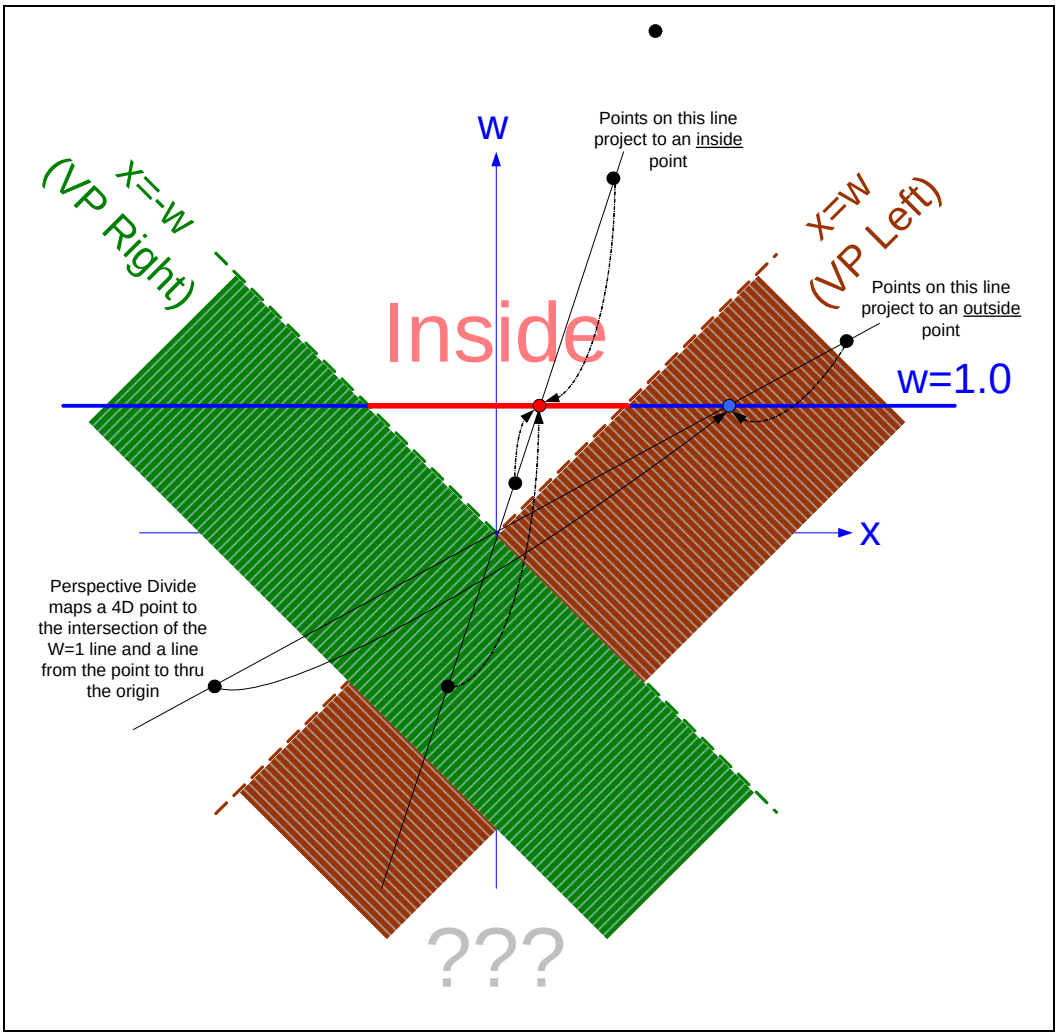
The CLIP unit will perform view volume clip test using NDC coordinates (the results of the speculative PerspectiveDivide). The treatment of negative ndc.w and invalid (NaN, +/-INF) coordinates is clarified below.

Negative W Coordinates

Consider for a moment vertices with a negative clip.w coordinate. Examination of the API definitions for "outside" shows that it is impossible for that vertex to be considered inside both the XMIN (NDC Left) and XMAX (NDC Right) planes. The clip.x coordinate would need to be greater than or equal to some positive value (-clip.w) to be considered inside the XMIN plane, while also being less than or equal to the negative (clip.w) value to be considered inside the XMAX plane. Obviously both these conditions cannot be met simultaneously, so a vertex with a negative clip.w coordinate will always appear outside.

Surprisingly, it is possible for a vertex to be outside both the XMIN and XMAX planes (and likewise for the Y axis). This arises when clip.w is negative and clip.x falls between clip.w and -clip.w. Note, however, that in NDC space (post perspective-divide), this same vertex would be considered inside. This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. The CLIP stage will avoid this artifact by supporting an additional clip.w=0 clip plane – a negative ndc.rhw value indicates the point is outside of the clip.w=0 plane. (See sections below for related errata in DevBW and DevCL devices)

The assumption made in the Clip stage is that only the w>0 portion of clip space is considered visible. The VertexClipTest function tests each incoming 1/w value and, if



negative, the vertex is tagged as being outside the w=0 plane. These vertex outcodes are combined in ClipDetermination to determine TA/TR/MC status.

A negative w coordinate poses an additional issue due to the fact that VertexClipTest is performed using post-perspection-projection coordinates (NDC or screen space). This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. For example, to test for (x>w) using NDC coordinates, (x/w>1) must be used when w>0, and (x/w<1) must be used when w<0. The VertexClipTest function therefore uses the sign of the incoming 1/w coordinate to select the appropriate comparison function for each of the VP and GB clip planes.

As the CLIP thread performs clipping in 4D clip space, only the truly visible portions of objects (i.e, meeting the 4D clip space visibility criteria) will be considered. The CLIP thread should not output negative w (clip or NDC) coordinates.

6.2.2 User-Specified Clipping

The various APIs define mechanisms by which objects can be clipped or culled according to some user-specified parameter(s) in addition to the implied viewport clipping. In GEN4, the HW support of these mechanisms is restricted to use of the 8 UserClipFlags (UCFs) of the VUE Vertex Header. Software is required to provide the remaining support (e.g., the JITTER including GEN4 instructions to cause a distance value to be computed, tested for visibility, and generation of the appropriate UCF bit.)

6.2.2.1 User Clip Planes

Up to 6 *user clip planes* can be defined and enabled. These planes define half-spaces that are intersected with the view volume (and each other) to form a final clip volume. Each user clip plane is specified by four coefficients of a plane equation in clip space coordinates (`UserClipPlane[n].xyzw`). A point is not visible if it has a negative distance to the plane. Therefore, points P that satisfy the following equation are considered to lie in the half-space and therefore may be visible:

$$(P.xyzw \text{ dot } UCP[n].xyzw) >= 0, \qquad 0<=n<=5$$

There is no direct HW support for this distance computation. The driver/JITTER is required to cause the distances to be correctly computed/compared in a shader, with the comparison result (Boolean) placed in the proper location in the Vertex Header.

6.2.3 Negative-W Clipping Errata

In DevBW and DevCL-A devices there is a bug in the definition of the handling of negative RHW (1/w) coordinates in the Clip unit's trivial reject logic. The fault may cause line and triangle objects to be erroneously trivially rejected and therefore be manifested as occasional missing geometry.

This section also describes a partial fix (ECO) that is incorporated into DevCL-B, and an additional ECO HW change for DevBW-E0.

DevCL-B ECO (partial fix)

The DevCL-B ECO parallels the PreDevBW-E0, DevCL-A SW workaround in that it uses UC7 logic to provide full trivial-accept (TA), trivial-reject (TR) and mustclip (MC) support for the w=0 clip plane. The pre-clipper shader kernel will have to be modified to set NDC x/w, y/w, z/w to 0.0 if w<0. However, this ECO allows all 8 UserClipFlags to be supported (with limitations).

Note that this ECO is suboptimal due to constraints on the location and extent of the ECO. A bit (ECOSKPD[9]) is included to revert back to the DevCL-A behavior; therefore, providing driver (SW-workaround) compatibility

DevBW-E0 ECO (partial fix)

This ECO extends the DevCL-B ECO described above. In VertexClipTest, if the vertex has a negative W coordinate, the VP & GB outcodes are inverted (if enabled). (In addition, a bug related to mis-handling of z = -0 is resolved, but that is unrelated to neg-w handling). Note that the inversion of the outcodes is not entirely correct in that

it mis-handles the '=' condition. As a result, the clip boundaries will be treated as "outside" in the negative-w regions. (Unfortunately correct handling the '=' case made the ECO untenable).

On the bright side, this ECO:

- Removes the need for any VS/GS software workaround. The HW will detect a negative w and compute the (almost-correct) VP & GB outcodes.
- Removes the need to set UserClipFlagsMustClipEnable. There is no reason to force a clip thread specifically for UC7 (which is set if w<0). As the outcodes are set correctly even when w<0, clip threads will be spawned as required. In addition, objects completely in w<0 space will be correctly TR'd against UC7 (assuming that UC is enabled).

However, UC7) will still be routed to the BAD outcode and subsequently will cause a clip thread to be spawned – therefore spawning clip threads for objects with any vertex having UC7 set.

The following table summarizes the software workarounds required for the various devices

Figure 6-1. SW Workaround Summary

Device	VS/GS Kernel	Clip State	Clip Kernel	Notes

DevBW
DevCL-A
	If (w<0) {
	npc.xyzw=0
	UC7=1
	}
		Enable UC7
		Set UCFMustClipEnable to force clips for mixed NEGW cases.
			If UC7 set, other outcodes are undefined and must be recomputed.	UC7 unavailable for normal use

DevCL-B+	If (w<0) {
	npc.xyz=0
	}
		Enable UC7
		Set UCFMustClipEnable to force clips for mixed NEGW cases.
			If UC7 set, other outcodes are undefined and must be recomputed.
			Will see “BAD” objects due to BAD←UC7 hack.
				UC7 is supported (routed to BAD before being used for NEGW).

DevBW-E0+	No WA required	Enable UC7 in order to allow TR against w<0. No need to set UCFMustClipEnable.	Will see “BAD” objects due to BAD←UC7 hack.	UC7 is supported (routed to BAD before being used for NEGW).

6.2.3.1 W Clipping Errata (DevBW, DevCL-A)

The DevBW and DevCL-A devices contain a definitional error in that a separate clip.w=0 clip plane was not implemented, and instead a negative ndc.rhw value caused all clip outcodes (except for BAD and UCs) to get set in VertexClipTest. This behavior can lead to false trivial rejects for line and triangle objects. The Trivial Reject function is therefore UNDEFINED under the following conditions:

1. Line or triangle object
2. At least one vertex has a negative RHW component
3. At least one vertex has a non-negative RHW component
4. All vertices straddle one or more common VP/GB clip planes
5. All vertices are not outside of a common enabled clip plane (including UCFs) – i.e., the object should not be trivially rejected

Software must prevent these conditions from occuring whenever it uses a Clip Mode which uses the trivial reject function (NORMAL or CLIP_NON_REJECTED). A suggested workaround is to have the previous shader (VS or GS) detect negative w coordinates, and if seen, set all NDC coordinates (x/w, y/w, z/w, 1/w) in the Vertex Header with 0.0, and set/utilize a UserClipFlag to cliptest against w=0.

6.2.3.2 W Clipping Errata (DevCL-B)

The DevCL-B device includes a partial fix for the errata (previous section). The fix parallels the suggested Dev-BW,Dev-CL-A SW workaround in that it uses UC7 logic to provide full trivial-accept (TA), trivial-reject (TR) and mustclip (MC) support for the w=0 clip plane (thus correctly handling negative RHW components). The pre-clipper shader kernel will have to be modified to set NDC x/w, y/w, z/w to 0.0 if w<0. However, this ECO allows all 8 UserClipFlags to be supported (with limitations) and therefore is applicable to a D3D10 driver.

A bit (ECOSKPD[9]) is included to revert back to the DevCL-A behavior, therefore providing driver (SW-workaround) compatibility (though posing an issue for D3D10 support).

The fix consists of 4 parts:

(1) Reroute UserClipFlag[7] into BAD

In VertexClipTest, instead of

```
outcode[BAD] = ISNAN(rhw)
```

the fix adds

```
outcode[BAD] = ISNAN(rhw) || UserClipFlag[7]
```

In concert with (4) (BAD Forces SPAWN, below), this change will force SPAWN whenever a vertex has rhw==NaN or has UserClipFlag[7] set, assuming REJECT_ALL mode is not in effect. Previously, only rhw==NAN lead to a BAD object, and all BAD objects were discarded except in CLIP_ALL mode. This change allows a D3D10 driver to use all 8 UserClipFlags for clipDistance and cullDistance, an improvement over the previous SW workaround. However, there are limitations and ramifications (see below).

(2) Prevent Setting of All Outcodes upon Negative RHW

In VertexClipTest, the fix removes the following logic (which was the source of the original problem):

```
—
—       if (0 rhw_neg)
—       {
    —   outCode[VP_XMIN] = 1
    —   outCode[VP_XMAX] = 1
    —   outCode[VP_YMIN] = 1
    —   outCode[VP_YMAX] = 1
    —   outCode[VP_ZMIN] = 1
    —   outCode[VP_ZMAX] = 1
    —   outCode[GB_XMIN] = 1
    —   outCode[GB_XMAX] = 1
    —   outCode[GB_YMIN] = 1
    —   outCode[GB_YMAX] = 1
    —   goto UserClipFlags
—       }
```

This change prevents some false trivial rejects. However, it is not a complete fix in that the computed VP,GB outcodes are still not correct when w<0. In order to completely remove false TRs, the pre-clipper kernel must set x/w, y/w and z/w (the NDC coordinates in the vertex header) to 0.0 whenever w<0. Note that 1/w must be passed normally (not forced to 0.0 as in the DevBW,DevCL-A workaround) – as the sign of 1/w is used to set UC7 (see below).

(3) Reroute rhw_neg into UCF7

In VertexClipTest, instead of

```
outcode[UC7] = UserClipFlag[7] && UserClipFlagClipTestEnable[7]
```

the fix adds

```
outcode[UC7] = rhw_neg && UserClipFlagClipTestEnable[7]
```

This change routes UserClipFlag[7] into BAD, thus using UC7 logic to perform TA/TR/MC determination for the w=0 clip plane. Note that UCFClipTestEnableMask[7] still applies to UC7, though UC7 is now sourced from rhw_neg instead of UserClipFlag[7].

(4) BAD Forces SPAWN except in REJECT_ALL Mode

In the application of ClipMode, a BAD object (any vertex has rhw=NaN or UserClipFlag[7] set) will force a SPAWN unless ClipMode is REJECT_ALL. This is what provides support for cliptest/clipping against UserClipFlag[7]. Performance-wise, neither of these BAD cases are expected to occur very often (at least compared to negative W).

6.2.3.2.1 Support for Clip-Testing Against W=0

Software must set UserClipFlagsClipTestEnable[7] to enable clip-testing against the w=0 plane. If set, the rhw_neg bit will be routed to UC7, therefore permitting trivial reject, trivial accept and must clip determination like the other seven UCFs.

As the rhw_neg bit is handled as a UCF, it is subject to the UserClipFlagsMustClipEnable state bit. If this state bit is set, UCFs (by themselves) can lead to a mustclip determination (for the assumed purpose of 3D clipping against that plane). This is the expected setting for D3D9 and OGL use of the UCFs. If clear, the UCFs (by themselves) will not lead to a mustclip determination. This is the expected setting for D3D10, where the cullDistance/clipDistance functions do not require 3D clipping (only accept/reject). Unfortunately, as 3D clipping against w=0 will be required and rhw_neg appears as a UCF, this effectively forces the D3D10 driver to **set** this state bit. This will likely cause some objects to be routed to a clip thread that otherwise could have been passed down the pipeline as a trivial accept (e.g., the object is inside of all clip planes except for straddling UserClipFlag[4]). The clip kernel can be modified to detect these cases and pass (copy) the input object as output.

6.2.3.2.2 Support for UserClipFlag[7]

This flag should only be required in support of the maximum complement of eight cullDistance/clipDistance values. When any vertex of an object has this bit set, the object will be sent to a clip thread (unless in REJECT_ALL mode when all objects are discarded). Note that since the object is considered "BAD", the ObjectOutcode[BAD] bit in the payload will be set. The clip kernel would need to examine each vertex's rhw value and UserClipFlag[7] bit to distinguish between the (now) two causes of BAD objects.

Trivial Reject: Unfortunately, the HW does not support trivial-reject against UserClipFlag[7]. Instead, the clip kernel can detect that a set UserClipFlag[7] was what caused the object to be considered BAD, and do an early discard of the object if all vertices had UserClipFlag[7] set. It is assumed that normal operation of the clip kernel would also lead to the discard of these objects, albeit in a less optimal fashion.

Clipping: Once the clip kernel determines that (a) the object was not BAD due to rhw==NaN, and (b) the object cannot be rejected against the UserClipFlag[7] bit, then it has determined that the object straddles the clip plane associated with UserClipFlag[7]. For D3D10 cullDistance, the API specifies that non-culled primitives are not subject to clipping. For D3D10 clipDistance, the API specifies that the clipping should be done at the pixel level (for GEN4.x, in the PS). In either case, the clip kernel can simply copy the input object to its output and send the object down the pipeline – no 3D clipping is required.

Enable: As opposed to the UserClipPlanes there are no state "enables" associated with cullDistance/clipDistance values – the app must match up the generation and PS use of these values in the shader declarations. Therefore the use of UserClipFlagsClipTestEnable[7] to control use of rhw_neg as UC7 should pose no direct issue to the driver. The shader which computes the cullDistance/clipDistance value associated with UserClipFlag[7] should only cause the UCF7 bit to be set when the appropriate criteria is met (i.e., the associated cullDistance/clipDistance value is negative or NaN).

6.2.4 Tristrip Clipping Errata [Pre-DevBW-E1], [DevCL]

The HW clip unit has an implementation bug in the ClipDetermination logic related to the processing of tristrip primitives (TRISTRIP and TRISTRIP_REV). If an object in the tristrip is determined to be a trivial reject case (TR), and the next object in the strip is determined to be a trivial accept (TA) case, a primitive topology can be emitted (for that TA object and possibly subsequent objects) with an incorrect primitive topology type. More specifically, instead of emitting a TRISTRIP_REV primtype, a TRISTRIP primtype may be omitted, and vice versa. This will lead to incorrect face culling (if enabled) downstream in the SF unit and be manifested by missing/extra triangles rendered.

TR to TA transitions can occur with when the tristrip crosses a viewport XY or UCF clip boundary. Note that this is not an issue with the VPZ or GBXY boundaries, as crossing one of those boundaries would cause at least one MustClip (MC) object between TR and TA objects and therefore the fault is not encountered. The same applies to VPXY

when the GBXY cliptest is disabled (as there objects will get clipped against the VPXY boundaries).

There is a way for software to work around this problem, assuming that avoiding the use of trsitrips in the first place is not practical. Software can disable cliptest against the VPXY (assuming GBXY is disabled) and UCF flags prior to submitting tristrip primitives. This will likely incur a performance penalty as objects that could be trivially rejected against these boundaries will be sent down the pipe. Note that objects that would have been TR-ed against VPXY will likely be discarded in the SF unit's 2D clipping logic, so only partial SF processing will be incurred.

The same is not true for the UCF flags. When used for "ClipDistance", the could-have-been-TRed objects will be completely set-up and rasterized, with the PS kernel eventually killing all pixels. When used for 'CullDistance", the feature will appear to be non-functionaly as no culling will occur. One way to avoid some of this performance penalty would be for software to to leave the **UCF ClipTest Enable Bitmask** bits set, but also set the **UserClipFlags MustClip Enable** bit. This would (a) permit trivial reject against the UCFs, and (b) avoid the fault condition by forcing a MustClip case between TR and TA objects. The clip kernel would simply need to pass through any UCF-clipped only objects (which should be the default operation of the clip kernel).

6.2.5 Guard Band

3DClipping is time consuming. For cases where 2DClipping is sufficient, we are willing to forgo 3DClipping and instead apply 2DClipping during rendering. In the general case, this is possible only when an object is totally within the ZMin and ZMax planes, and only clipping to the view volume X/Y MIN/MAX clip planes is required, as 2DClipping is restricted to a screen-aligned 2D rectangle.

However, we must ensure that the 2D extent of these objects do not exceed the limitations of the renderer's coordinate space (see Vertex X,Y Clamping and Quantization in the SF section). Therefore we define a 2D *guardband* region corresponding to (though likely somewhat smaller than) the maximum 2D extent supported by the renderer. During VertexClipTest, vertices are (optionally) subjected to an additional visibility test based on the 2D guardband region.

During ClipDetermination, if an object is not trivially-rejected from the 2D viewport, the XMIN_GB, XMAX_GB, YMIN_GB and YMAX_GB guardband outcodes are used instead of the XMIN, XMAX, YMIN, YMAX view volume outcodes to determine trivial-accept. This will allow objects that fall within the guardband and possibly intersect the viewport to be trivially-accepted and passed down the pipeline.

The diagram below shows some examples of objects (triangles) in relation to the viewport and guardband. The shaded triangles are examples of triangles that are not trivially accepted to the viewport but trivially accepted to the guardband and therefore passed to down the pipeline. Without the guardband, these triangles would have to be submitted to a CLIP thread.

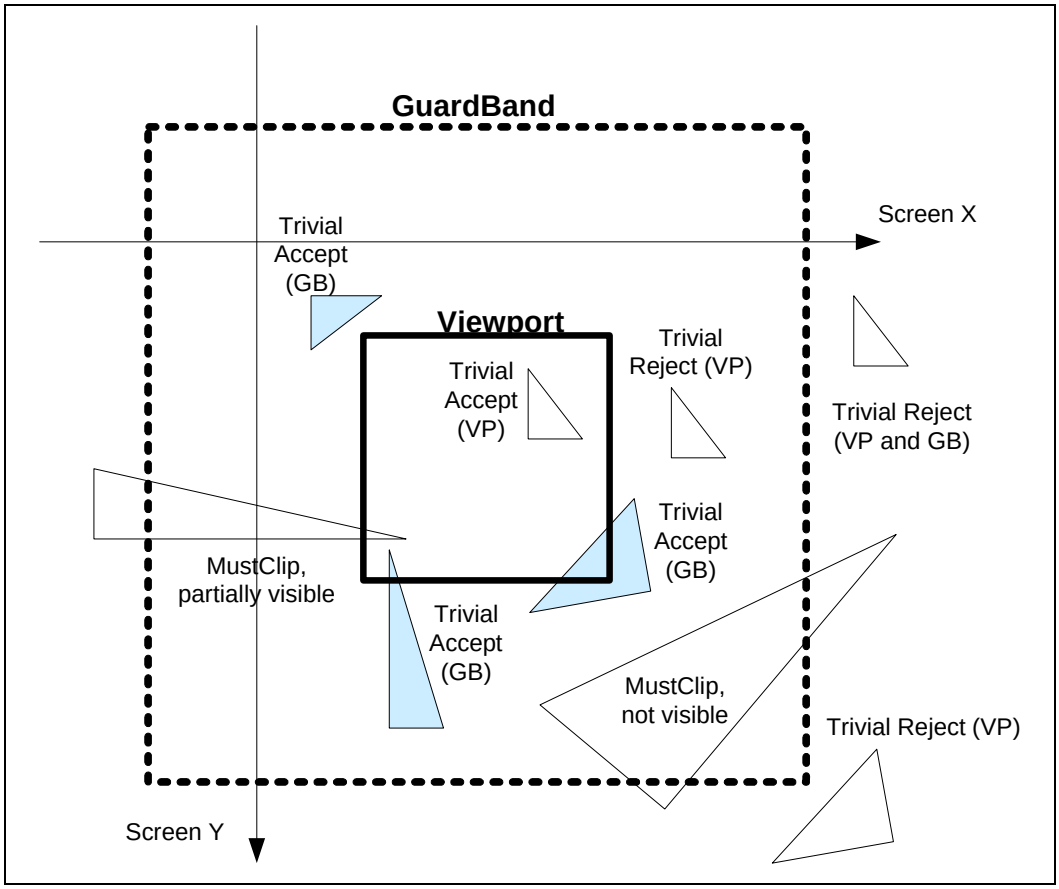


Figure 6-2. Normal Guardband Operation

The CLIP stage needs to handle the case where the viewport XY is larger than the screen space coordinate range supported by the SF and WM units. This condition may arise when the API defines an implicit 2D clip between the viewport XY extent and the rendertarget. In the GEN4 3D pipeline, the guardband must used to force explicit clipping in order to ensure legal coordinates are passed out of the CLIP stage. Therefore the CLIP unit supports a guardband that can be larger or smaller than the viewport (in any particular direction). The following diagram illustrates a case with a very large viewport, extending well beyond the guardband. Note that the only trivial accept case is where objects are completely within the guardband.

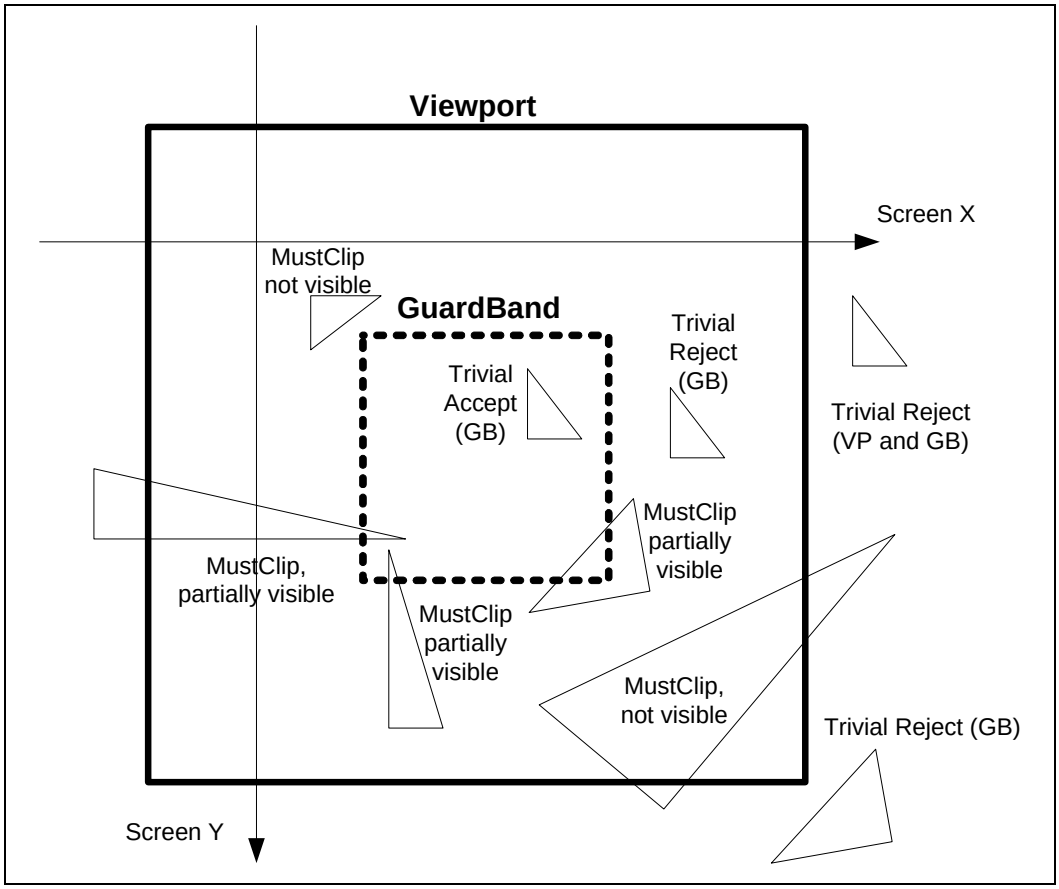


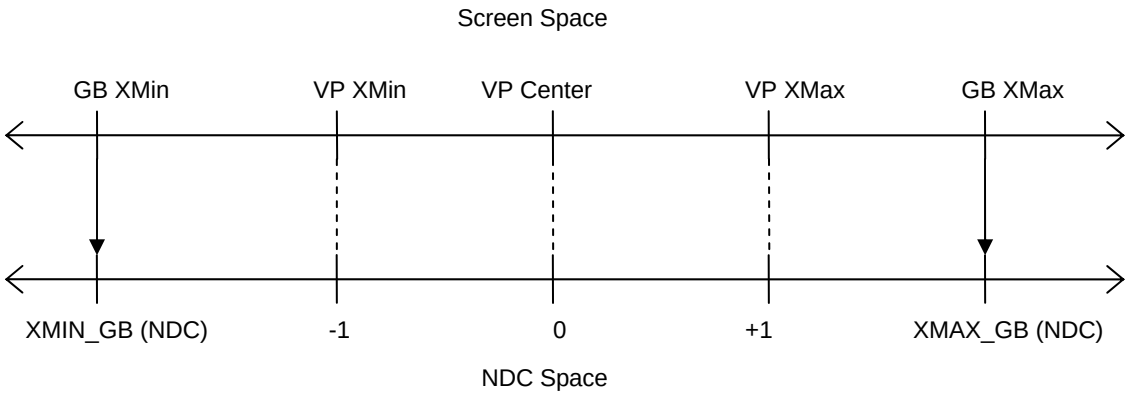
Figure 6-3. Very Large Viewport Case

Programming Restriction: Varying ViewportIndex within Strip-based Primitives

The forementioned case, where objects must be clip-tested and clipped against the guardband, leads to an somewhat obscure CLIP unit programming restriction. The fundamental issue is that the CLIP unit does not natively support clip-testing of strip topologies where the ViewportIndex can vary from vertex to vertex. The proper handling of this (i.e., applying the ViewportIndex from the leading vertex of the object to all object vertices) would require clip-testing on a per-object, not a per-vertex, basis. As the CLIP unit only uses the ViewportIndex to access the corresponding viewport-normalized guardband parameters, this exceptional condition could be ignored by turning off the guardband and thereby ignoring the incorrect results provided by the guardband cliptest. (Note that the viewport cliptest is performed against fixed values and therefore not dependent on the ViewportIndex). This leads to the conflict where the guardband needs to be enabled (to handle a very large guardband) but disabled (to ignore the incorrect cliptest results for strips with varying ViewportIndex). In this case, software will likely have to resort to use of the CLIP_ALL Clip Mode. This will pass all objects to a CLIP thread, where the correct clip-testing and clipping can be performed.

6.2.5.1 NDC Guardband Parameters

When the CLIP unit performs VertexClipTest in NDC space, the guardband limits must be provided as NDC coordinates. The diagram below shows how the guardband NDC coordinates are derived. Specifically, the XMIN_GB NDC coordinate is simply the ratio of the (screen space) distance from the screen space VP center to the screen space GB XMin boundary over the distance from the VP center to the VP XMin (left) boundary. A similar computation yields the XMAX_GB (right), YMIN_GB (bottom) and YMAX_GB (top) guardband NDC coordinates.



As these guardband parameters are defined relative to the viewport, each of the up-to-16 sets of viewport specifications supported in the 3D pipeline will require a corresponding set of guardband parameters. These guardband parameters are provided as a separate memory-resident state structure (CLIP_VIEWPORT), and referenced via the **Clipper Viewport State Pointer** contained in the CLIP_STATE structure. Note that the CLIP_VIEWPORT structure has a different definition than the SF_VIEWPORT structure used by the SF unit.

6.2.5.2 Screen Space Guardband Parameters

When the CLIP unit performs VertexClipTest in screen space, the guardband limits must be provided as screen space coordinates. Note that YMIN_GB will correspond to the screen space GB top, and YMAX_GB will correspond to the screen space GB bottom, which is opposite from the NDC case.

6.2.6 Vertex-Based Clip Testing & Considerations

The CLIP unit performs clip test and determines whether objects need to be clipped based solely on information (position, UserClipFlags) provided at the vertices of the object as they arrive at the clip stage. Issues arise if and when the corresponding rendered object is not constrained to the convex hull of the object. Different APIs impose different treatment of these conditions.

In addition and in the more general case, a CLIP thread could be used to convert the object (as defined by its vertices) into some arbitrary output primitive. In this case, the CLIP unit's ClipTest/ClipDetermination logic may not be suitable for determination of when to reject/accept/clip objects. In this case the ClipMode can be used to route all (or all non-rejected) objects to CLIP threads, where the proper clip-test and clipping can occur in the CLIP kernel.

One issue that arises is whether a trivial-reject to the VPXY is suitable. If this were allowed, an object might be discarded even if it would have been partially visible in the viewport. A second issue is whether a TA against the GB is suitable. If this were allowed, portions of the rendered object might be visible in the VP even if the object should have been clipped out of the VP.

6.2.6.1 Triangle Objects

In the normal processing of triangle-based primitives (tristrip/trilist/polygon/etc.), the footprint of each triangle is constrained to the 2D convex hull. I.e., the rendering of these triangles will not produce pixels outside of the triangle. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for triangle objects:

- Both the VPXY and GB clip boundaries can be utilized (as described above). If the triangle is TR against the VP, it can be discarded. Otherwise, if the triangle is TA against the GB, it can be passed down the pipeline (assuming it is TA against VPZ, UCFs, etc.) and properly handled by 2DClipping.
- The GB parameters can be programmed to coincide with the maximum allowable screen space extent (though making the GB marginally smaller than this max extent is highly recommended).

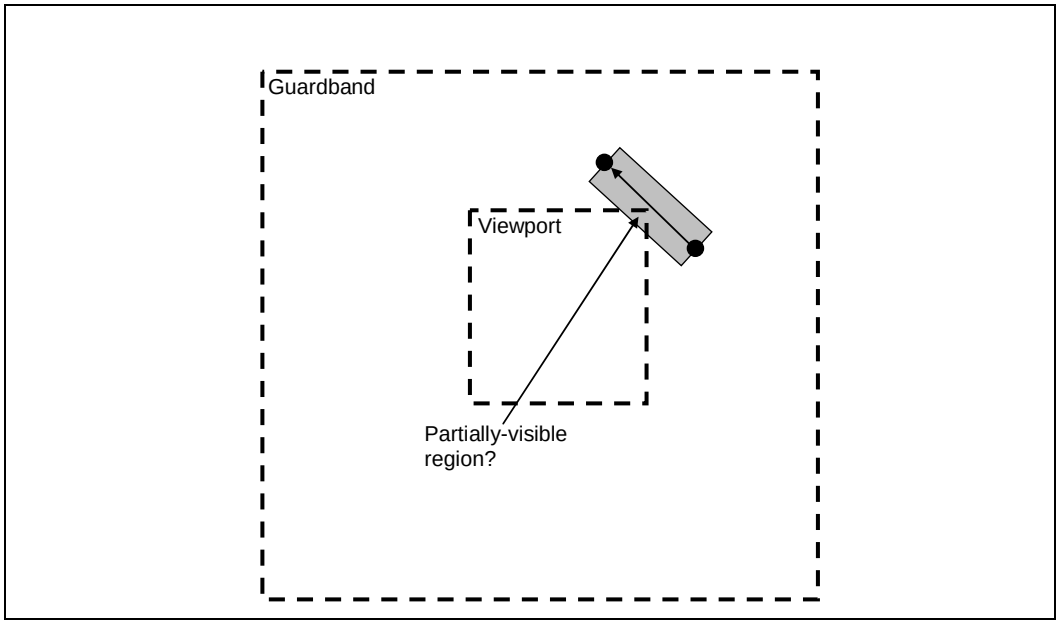
6.2.6.2 Non-Wide Line Objects

In the normal processing of non-wide, line-based primitives (linestrip/linelist/etc.), the footprint of each line is constrained to the 2D convex hull. I.e., the rendering of these lines will not produce pixels off of the line. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for non-wide line objects. (See Triangle Objects above).

6.2.6.3 Wide Line Objects

The GEN4 rendering hardware supports wide lines (solid lines with a line width or anti-aliased lines). When rendered, pixels outside of the convex hull will be generated.

The following diagram shows an example of a wide line that normally would be TA against the GB. If the TA is allowed, the partially-visible region of the line would be rendered.

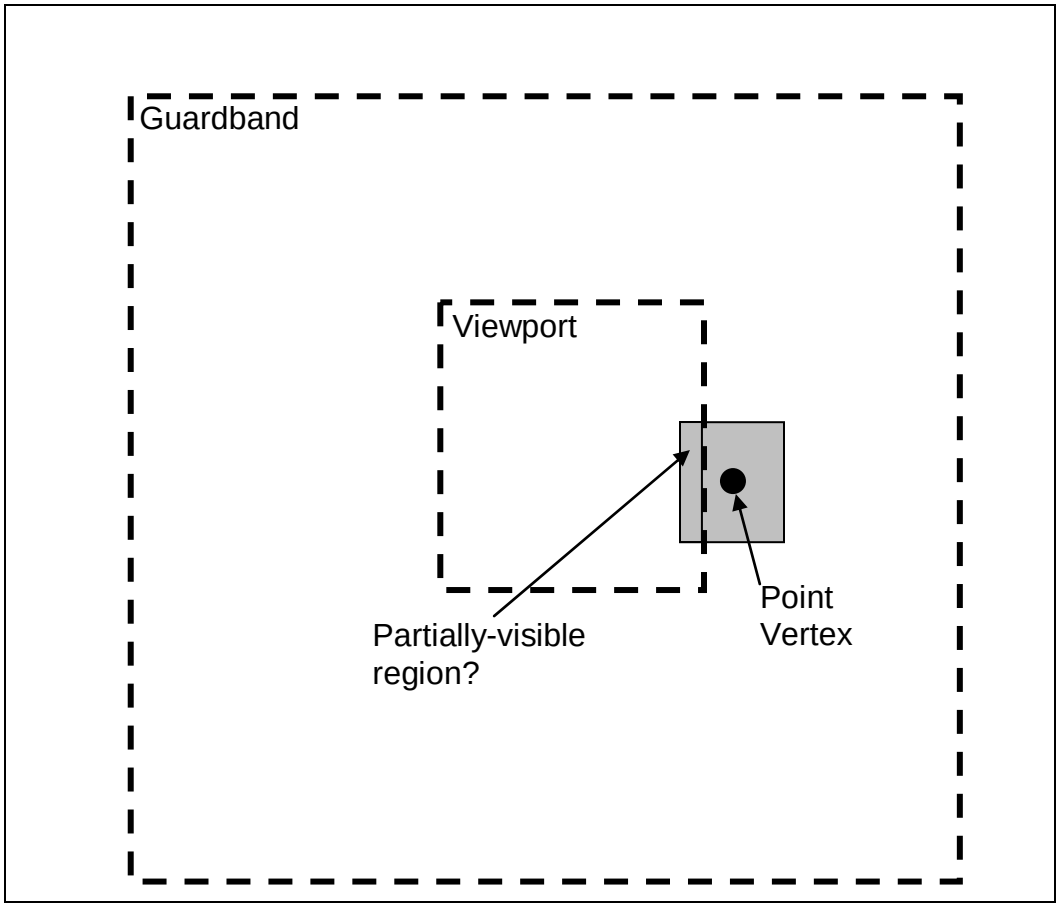


In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the line must be clipped-out against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide lines.

6.2.6.4 Wide Points

The GEN4 rendering hardware supports a width parameter for native line objects. When rendered, pixels surrounding the point (center) vertex will be generated.

The following diagram shows an example wide point that normally would be TR against the VPXY. If the TR is allowed, the partially-visible region of the point would not be rendered.



In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the point must be TR against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide points.

6.2.6.5 RECTLIST

The CLIP unit treats RECTLIST exactly like TRILIST. No special consideration is made for the implied 4th vertex of each rectangle (although ViewportXY and Guardband VertexClipTest theoretically should be sufficient to drive ClipDetermination). Given this, and the fact that RECTLIST is primarily intended for driver-generated "BLT" functions, there are number of restrictions on the use of RECTLIST, especially regarding the CLIP unit. Refer to the RECTLIST definition in 3D Pipeline.

6.2.7 3D Clipping

If an object needs to be clipped, it will be passed to the CLIP thread. The CLIP thread will perform some (arbitrary) algorithm to clip the primitive, and subsequently output "new" vertices as a primitive defining the visible region of the input object (assuming there is a visible region). In the process of spawning the CLIP thread, the input vertices may be considered "consumed" and therefore dereferenced. Therefore the

CLIP thread will need to copy (if required) any input VUE data to a new output VUE – there is no mechanism to “output” input vertices other than copying.

6.3 CLIP Stage Input

As a stage of the GEN4 3D pipeline, the CLIP stage receives inputs from the previous (GS) stage. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the CLIP stage.

6.3.1 State

6.3.1.1 CLIP_STATE

The following table describes the format and contents of the CLIP_STATE structure referenced by the **Pointer to CLIP State** field of the 3DSTATE_PIPELINED_POINTERS command.

CLIP_STATE		
Project:	All	
Controls the CLIP stage hardware.		
DWord	**Bit**	**Description**
0	31:6	**Kernel Start Pointer** Project: All Format: GeneralStateOffset[31:6] This field specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address** **[DevBW-A,B] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)
	5:4	**Reserved** Project: All Format: MBZ
	3:1	**GRF Register Count** Project: All Format: U3 register block count - 1 Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16.
	0	**Reserved** Project: All Format: MBZ

CLIP_STATE

1	31	**Single Program Flow (SPF)** Project: All Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). **Value** **Name** **Description** **Project** 0h Reserved All 1h Enable Single Program Flow enabled All
	30:26	**Reserved** Project: All Format: MBZ
	25:18	**Binding Table Entry Count** Project: All Format: U8 Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache.
	17	**Thread Priority** Project: All Specifies the priority of the thread for dispatch **Value** **Name** **Description** **Project** 0h Normal Priority All
	16	**Floating Point Mode** Project: All Specifies the initial floating point mode used by the dispatched thread. **Value** **Name** **Description** **Project** 0h Use IEEE-754 Rules All 1h Use alternate rules All
	15:14	**Reserved** Project: All Format: MBZ
	13	**Illegal Opcode Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	12	**Reserved** Project: All Format: MBZ
	11	**Mask Stack Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*.
	10:8	**Reserved** Project: All Format: MBZ

CLIP_STATE		
	7	**Software Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	6:0	**Reserved** Project: All Format: MBZ
2	31:10	**Scratch Space Base Pointer** Project: All Format: GeneralStateOffset[31:10] Specifies the location of the scratch space area allocated to this FF unit, specified as a 1KB-granular offset from the **General State Base Address**. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space.**
	9:4	**Reserved** Project: All Format: MBZ
	3:0	**Per-Thread Scratch Space** Project: All Format: U4 power of 2 Bytes over 1K Bytes FormatDesc Range [0,11] indicating [1K Bytes, 2M Bytes] Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.
3	31	**Reserved** Project: All Format: MBZ
	30:25	**Constant URB Entry Read Length** Project: All Format: U6 FormatDesc Range [0,63] Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments.
	24	**Reserved** Project: All Format: MBZ
	23:18	**Constant URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload.
	17	**Reserved** Project: All Format: MBZ

CLIP_STATE

	16:11	**Vertex URB Entry Read Length** Project: All Format: U6 FormatDesc Range [1,63] Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. **Programming Notes** It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.
	10	**Reserved** Project: All Format: MBZ
	9:4	**Vertex URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread.
	3:0	**Dispatch GRF Start Register for URB Data** Project: All Format: U4 FormatDesc Range [0,15] indicating GRF [R0,R15] Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload.
4	31:30	**Reserved** Project: All Format: MBZ
	29:25	**Maximum Number of Threads** Project: All Format: U5 thread count – 1 Range [0,1] indicating thread count of [1,2] Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. **Programming Notes** — **Project** When running in dual-thread mode, the **Number of URB Entries** field must contain an even number. Each thread will be allocated one half the total number of entries. — All A URB_FENCE command must be issued subsequent to any change to the value in this field (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See *Graphics Processing Engine* (Command Ordering Rules) — All

CLIP_STATE

	24	**Reserved** Project: All Format: MBZ
	23:19	**URB Entry Allocation Size** Project: All Format: U5 count (of 512-bit units) – 1 Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows Specifies the length of each URB entry owned by this FF unit. **Programming Notes** — **Project** Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All
	18	**Reserved** Project: All Format: MBZ
	17:11	**Number of URB Entries** Project: All Format: U7 Count of URB entries Range [1,32] if GS enabled, otherwise ignored. Specifies the number of URB entries that are used by this FF unit. **Programming Notes** — **Project** When running in dual-thread mode, the **Number of URB Entries** field must contain an even number. Each thread will be allocated one half the total number of entries. — All If ENABLED, the GS stage must be allocated at least one URB entry — All Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All
	10:0	**Reserved** Project: All Format: MBZ
5	31:30	**Reserved** Project: All Format: MBZ
	29	**Vertex Position Space** Project: All This field specifies the coordinate system within which the incoming Vertex Position X,Y,Z values are defined. The setting affects VertexClipTest.

Value	Name	Description	Project
0h	VPOS_NDCSPACE	Vertex Position is in NDC space	All
1h	VPOS_SCREENSPACE	Vertex Position is in Screen space	All

CLIP_STATE		
	28	**Viewport XY ClipTest Enable** Project: All Format: Enable This field is used to control whether the Viewport X,Y extents are considered in VertexClipTest. See Tristrip Clipping Errata subsection.
	27	**Viewport Z ClipTest Enable** Project: All Format: Enable This field is used to control whether the Viewport Z extents (near, far) are considered in VertexClipTest.
	26	**Guardband ClipTest Enable** Project: All Format: Enable This field is used to control whether the Guardband X,Y extents are considered in VertexClipTest for non-point objects. If the Guardband ClipTest is DISABLED but the Viewport XY ClipTest is ENABLED, ClipDetermination operates as if the Guardband were coincident with the Viewport. If both the Guardband and Viewport XY ClipTest are DISABLED, all vertices are considered “visible” with respect to the XY directions.
	25	Reserved: MBZ
	24	**UserClipFlags MustClip Enable** Project: All Format: Enable This field is used to include the UserClipFlags in MustClip determination, in order to support clipping to User Clip Planes. If ENABLED, the setting of enabled UserClipFlag bits can cause a CLIP thread to be spawned. If the enabled UCF values at the object vertices do not indicate a trivial accept or reject with relation to the UCFs, then a CLIP thread will be spawned (unless the object is trivially rejected for other reasons). If DISABLED, the UserClipFlags are only used for trivial accept or reject determination, and will not lead to a CLIP thread being spawned unless indicated by other cliptest results (or SV bits).
	23:16	**UserClipFlags ClipTest Enable Bitmask** Project: All Format: Enable This field is used to include the UserClipFlags in MustClip determination, in order to support clipping to User Clip Planes. If ENABLED, the setting of enabled UserClipFlag bits can cause a CLIP thread to be spawned. If the enabled UCF values at the object vertices do not indicate a trivial accept or reject with relation to the UCFs, then a CLIP thread will be spawned (unless the object is trivially rejected for other reasons). If DISABLED, the UserClipFlags are only used for trivial accept or reject determination, and will not lead to a CLIP thread being spawned unless indicated by other cliptest results (or SV bits).

CLIP_STATE

	15:13	**Clip Mode** Project: All This field specifies a general mode of the CLIP unit, when the CLIP unit is ENABLED. **Value** / **Name** / **Description** / **Project** 0h / CLIPMODE_NORMAL / TrivialAccept objects are passed down the pipeline, MustClip objects are passed to CLIP threads, TrivialReject and BAD objects are discarded / All 1h / CLIPMODE_ALL / All objects (including BAD objects & TrivReject) are passed to CLIP threads, regardless of classification / All 2h / CLIPMODE_CLIP_NON_REJECTED / TrivialAccept and MustClip objects are passed to CLIP threads, TrivReject and BAD objects are discarded / All 3h / CLIPMODE_REJECT_ALL / All objects are discarded / All 4h / CLIPMODE_ACCEPT_ALL / All objects (except BAD objects) are trivially accepted. This effectively disables the clip-test/clip-determination function. / All 5h / CLIPMODE_NORMAL_FFCLIP / Reserved / All 6h-7h / / Reserved / All **Errata** / **Description** / **Project** # / See previous sections (W Clipping Errata) for the description of errata regarding negative W and trivial reject. These errata impact the programming of Clip Mode. / DevBW, DevCL-A, DevCL-B
	12:0	**Reserved** Project: All Format: MBZ
6	31:5	**Clipper Viewport State Pointer** Project: All Format: GeneralStateOffset[31:5] Specifies the location of the current CLIP_VIEWPORT data structure, as a 32-byte aligned offset from **General State Base Pointer**). The CLIP unit accesses the viewport state throught it's Instruction/State Cache (ISC).
	4:0	**Reserved** Project: All Format: MBZ
7	31:0	**Screen Space Viewport X Min** Project: All Format: FLOAT32 This field contains the XMin (left) extent of the screen-space viewport. This field is only used when **Vertex Position Space** = VPOS_SCREENSPACE.

CLIP_STATE		
8	31:0	**Screen Space Viewport X Max** Project: All Format: FLOAT32 This field contains the XMax (right) extent of the screen-space viewport. This field is only used when **Vertex Position Space** = VPOS_SCREENSPACE.
9	31:0	**Screen Space Viewport Y Min** Project: All Format: FLOAT32 This field contains the YMin (top) extent of the screen-space viewport. This field is only used when **Vertex Position Space** = VPOS_SCREENSPACE.
10	31:0	**Screen Space Viewport Y Max** Project: All Format: FLOAT32 This field contains the YMax (bottom) extent of the screen-space viewport. This field is only used when **Vertex Position Space** = VPOS_SCREENSPACE.

6.3.1.2 CLIP_VIEWPORT

The viewport-related 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 viewport state array is aligned to a 32-byte boundary, and is located at (**General State Base Pointer** + **Clipper Viewport State Pointer)**.

Note that the definition of the CLIP_VIEWPORT structure differs from the SF_VIEWPORT structure used by the SF unit.

CLIP_VIEWPORT

Project: All

Viewport data used by the Clip unit.

DWord	Bit	Description
0	31:0	**XMin Clip Guardband** Project: All Format: FLOAT32 For VPOS_NDCSPACE: This 32-bit float represents the XMin guardband boundary (normalized to Viewport.XMin == -1.0f). This corresponds to the left boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the XMin guardband boundary in screen space coordinates. This corresponds to the left boundary of the screen space guardband.
1	31:0	**XMax Clip Guardband** Project: All Format: FLOAT32 For VPOS_NDCSPACE: This 32-bit float represents the XMax guardband boundary (normalized to Viewport.XMax == 1.0f). This corresponds to the right boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the XMax guardband boundary in screen space coordinates. This corresponds to the right boundary of the screen space guardband.
2	31:0	**YMin Clip Guardband** Project: All Format: FLOAT32 For VPOS_NDCSPACE: This 32-bit float represents the YMin guardband boundary (normalized to Viewport.YMin == -1.0f). This corresponds to the bottom boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the YMin guardband boundary in screen space coordinates. This corresponds to the top boundary of the screen space guardband.
3	31:0	**YMax Clip Guardband** Project: All Format: FLOAT32 For VPOS_NDCSPACE: This 32-bit float represents the YMax guardband boundary (normalized to Viewport.YMax == 1.0f). This corresponds to the top boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the YMax guardband boundary in screen space coordinates. This corresponds to the bottom boundary of the screen space guardband.

6.4 VertexClipTest Function

The VertexClipTest function compares each incoming vertex position (x,y,z,w) with various viewport and guardband parameters (either hard-coded values or specified by state variables).

The RHW component of the incoming vertex position is tested for NaN value, and if a NaN is detected, the vertex is marked as “BAD” by setting the outcode[BAD]. In general, any object containing a BAD vertex will be discarded, as (a) how to clip/render such objects is undefined, and (b) D3D10 specifies that such objects are to be silently discarded. However, in the case of CLIP_ALL mode, a CLIP thread will be spawned even for objects with “bad” RHW components. The CLIP kernel is required to handle this case, and can examine the **Object Outcode [BAD]** payload bit to detect the condition. (Note that the VP and GB Object Outcodes are UNDEFINED when BAD is set).

If the incoming RHW coordinate is negative (including negative 0) the NEGW outcode is set. Also, this condition is used to select the proper comparison functions for the VP and GB outcode tests (below).

Next, the VPXY and GB outcodes are computed, depending on the corresponding enable SV bits. If one of VPXY or GB is disabled, the enabled set of outcodes are copied to the disabled set of outcodes. This effectively defines the disabled boundaries to coincide with the enabled boundaries (i.e., disabling the GB is just like setting it to the VPXY values, and vice versa.).

The VPZ outcode is computed as required by the API mode SV.

Finally, the incoming UserClipFlags are masked and copied to corresponding outcodes.

The following algorithm is used by VertexClipTest:

```
—
—       //
—       // Vertex ClipTest
—       //
—       //  On input:
—       //  if (CLIP.PreMapped)
—       //    x,y are viewport mapped
—       //    z is NDC ([0,1] is visible)
—       //  else
—       //    x,y,z are NDC (post-perspective divide)
—       //  w is always 1/w
—
—       //
—       // Initialize outCodes to “inside”
—       //
—       outCode[*] = 0
—
—       //
—       // Check if w is NaN
—       // Any object containing one of these “bad” vertices
—       // will likely be discarded
—           //
```

```
                #ifdef (DevBW-E0 || DevCL-B)
                if (ISNAN(w)|| UserClipFlag[7])
                #else
                if (ISNAN(w))
                #endif
    {
       outCode[BAD] = 1
    }

    //
    //  If 1/w is negative, w is negative and therefore
    //  outside of the w=0 plane
    //
    //
    rhw_neg = ISNEG(rhw)
    if (rhw_neg)
    {
    #ifdef (PreDevBW-E0 || DevCL-A)
       outCode[VP_XMIN] = 1
       outCode[VP_XMAX] = 1
       outCode[VP_YMIN] = 1
       outCode[VP_YMAX] = 1
       outCode[VP_ZMIN] = 1
       outCode[VP_ZMAX] = 1
       outCode[GB_XMIN] = 1
       outCode[GB_XMAX] = 1
       outCode[GB_YMIN] = 1
       outCode[GB_YMAX] = 1
       goto UserClipFlags
 #endif
    }

    //
    // View Volume Clip Test
    //   If Premapped, the 2D viewport is defined in screen
    //      space
    //     otherwise the canonical NDC viewvolume applies
    //     ([-1,1])
    //
    if (CLIP_STATE.PreMapped)
    {
       vp_XMIN = CLIP_STATE.VP_XMIN
       vp_XMAX = CLIP_STATE.VP_XMAX
       vp_YMIN = CLIP_STATE.VP_YMIN
       vp_YMAX = CLIP_STATE.VP_YMAX
    } else {
       vp_XMIN = -1.0f
       vp_XMAX = +1.0f
       vp_YMIN = -1.0f
       vp_YMAX = +1.0f
    }
                if (CLIP_STATE.ViewportXYClipTestEnable) {
```

```
        —   outCode[VP_XMIN] = (x < vp_XMIN)
        —   outCode[VP_XMAX] = (x > vp_XMAX)
        —   outCode[VP_YMIN] = (y < vp_YMIN)
        —   outCode[VP_YMAX] = (y > vp_YMAX)
—                #ifdef (DevBW-E0)
—                  if (rhw_neg) {
        —       outCode[VP_XMIN] = (x >= vp_XMIN)
        —       outCode[VP_XMAX] = (x <= vp_XMAX)
        —       outCode[VP_YMIN] = (y >= vp_XMIN)
        —       outCode[VP_YMAX] = (y <= vp_XMAX)
        —   }
—                #endif
—                #endif
—                }
—                if (CLIP_STATE.ViewportZClipTestEnable) {
—                  if (CLIP_STATE.APIMode == APIMODE_D3D) {
        —       vp_ZMIN = 0.0f
        —       vp_ZMAX = 1.0f
—                  } else {  // OGL
        —       vp_ZMIN = -1.0f
        —       vp_ZMAX = 1.0f
—                  }
—                  outCode[VP_ZMIN] = (z < vp_ZMIN)
—                  outCode[VP_ZMAX] = (z > vp_ZMAX)
—                 #ifdef (DevBW-E0)
—                   if (rhw_neg) {
        —       outCode[VP_ZMIN] = (z >= vp_ZMIN)
        —       outCode[VP_ZMAX] = (z <= vp_ZMAX)
        —   }
—                #endif
—
    —       //
    —       // Guardband Clip Test
    —       //
    —       if {CLIP_STATE.GuardbandClipTestEnable) {
—                   gb_XMIN = CLIP_STATE.Viewport[vpindex].GB_XMIN
—                   gb_XMAX = CLIP_STATE.Viewport[vpindex].GB_XMAX
—                   gb_YMIN = CLIP_STATE.Viewport[vpindex].GB_YMIN
—                   gb_YMAX = CLIP_STATE.Viewport[vpindex].GB_YMAX
    —           outCode[GB_XMIN] = (x < gb_XMIN)
    —           outCode[GB_XMAX] = (x > gb_XMAX)
    —           outCode[GB_YMIN] = (y < gb_YMIN)
    —           outCode[GB_YMAX] = (y > gb_YMAX)
—                #ifdef (DevBW-E0)
—                  if (rhw_neg) {
        —       outCode[GB_XMIN] = (x >= gb_XMIN)
        —       outCode[GB_XMAX] = (x <= gb_XMAX)
        —       outCode[GB_YMIN] = (y >= gb_YMIN)
        —       outCode[GB_YMAX] = (y <= gb_YMAX)
        —   }
—                #endif
    —
```

```
   —      //
   —      // Handle case where either VP or GB disabled (but not
   —      // both)
   —      // In this case, the disabled set take on the outcodes
   —      // of the enabled set
   —      //
   —      if (CLIP_STATE.ViewportXYClipTestEnable &&
!CLIP_STATE.GuardbandClipTestEnable) {
   —          outCode[GB_XMIN] = outCode[VP_XMIN]
   —          outCode[GB_XMAX] = outCode[VP_XMAX]
   —          outCode[GB_YMIN] = outCode[VP_YMIN]
   —          outCode[GB_YMAX] = outCode[VP_YMAX]
   —      } else if (!CLIP_STATE.ViewportXYClipTestEnable &&
CLIP_STATE.GuardbandClipTestEnable) {
   —          outCode[VP_XMIN] = outCode[GB_XMIN]
   —          outCode[VP_XMAX] = outCode[GB_XMAX]
   —          outCode[VP_YMIN] = outCode[GB_YMIN]
   —          outCode[VP_YMAX] = outCode[GB_YMAX]
   —      }
   —
   —      //
   —      // X/Y/Z NaN Handling
   —      //
   —      xyorgben = (CLIP_STATE.ViewportXYClipTestEnable ||
CLIP_STATE.GuardbandClipTestEnable)
   —      if (isNAN(x)) {
   —          outCode[GB_XMIN] = xyorgben
   —          outCode[GB_XMAX] = xyorgben
   —           outCode[VP_XMIN] = xyorgben
   —           outCode[VP_XMAX] = xyorgben
   —       }
   —
   —      if (isNAN(y)) {
   —          outCode[GB_YMIN] = xyorgben
   —          outCode[GB_YMAX] = xyorgben
   —           outCode[VP_YMIN] = xyorgben
   —           outCode[VP_YMAX] = xyorgben
   —      }
   —
   —      if (isNaN) {
   —          outCode[VP_ZMIN] =
CLIP_STATE.ViewportZClipTestEnable
   —          outCode[VP_ZMAX] =
CLIP_STATE.ViewportZClipTestEnable
   —      }
   —
   —      //
   —      // UserClipFlags
   —      //
   —      ExamineUCFs
   —      for (i=0; i<7; i++)
   —      {
```

```
      —  outCode[UC0+i] = userClipFlag[i] &
CLIP_STATE.UserClipFlagsClipTestEnableBitmask[i]
—             }
—             #ifdef (DevBW-E0 || DevCL-B)
      —  outCode[UC7] = rhw_neg &
CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
—             #else
      —  outCode[UC7] = userClipFlag[i] &
CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
—     #endif
—
   —
```

6.5 Object Staging

The CLIP unit's Object Staging Buffer (OSB) accepts streams of input vertex information packets, along with each vertex's VertexClipTest result (outCode). This information is buffered until a complete object or the last vertex of the primitive topology is received. The OSB then performs the ClipDetermination function on the object vertices, and takes the actions required by the results of that function.

6.5.1 Partial Object Removal

The OSB is responsible for removing incomplete LINESTRIP and TRISTRIP objects that it may receive from the preceding stage (GS). Partial object removal is not supported for other primitive types due to either (a) the GS is not permitted to output those primitive types, and the VF unit will have removed the partial objects as part of 3DPRIMITIVE processing, or (b) although the GS thread is allowed to output the primitive type (e.g., LINELIST), it is assumed that the GS kernel will be correctly implemented to avoid outputting partial objects (or pipeline behavior is UNDEFINED). In short, CLIP unit partial object removal is only provided for the cases where the D3D10 GS shader programmer is able to generate partial objects.

An object is considered 'partial' if the last vertex of the primitive topology is encountered (i.e., PrimEnd is set) before a complete set of vertices for that object have been received. Given that only LINESTRIP and TRISTRIP primitive types are subject to CLIP unit partial object removal, the only supported cases of partial objects are 1-vertex LINESTRIPs and 1 or 2-vertex TRISTRIPs.

Partial Object Removal is performed only when the CLIP stage is ENABLED. If there is a possibility that the GS kernel can output incomplete objects, the CLIP stage must be ENABLED (as the SF stage does not tolerate incomplete objects). This may lead to a case where the CLIP stage needs to be ENABLED only to perform Partial Object Removal. In this case, if clipping is not desired, the Clip Mode may be set to ACCEPT_ALL.

6.5.2 ClipDetermination Function

In ClipDetermination, the vertex outcodes of the primitive are combined in order to determine the clip status of the object (TR: trivially reject; TA: trivial accept; MC: must clip; BAD: invalid coordinate). Only those vertices included in the object are examined (3 vertices for a triangle, 2 for a line, and 1 for a point). The outcode bit arrays for the vertices are separately ANDed (intersection) and ORed (union) together (across vertices, not within the array) to yield objANDCode and objORCode bit arrays.

TR/TA against interesting boundary subsets are then computed. The TR status is computed as the logical OR of the appropriate objANDCode bits, as the vertices need only be outside of one common boundary to be trivially rejected. The TA status is computed as the logical NOR of the appropriate objORCode bits, as any vertex being outside of any of the boundaries prevents the object from being trivially accepted.

If any vertex contains a BAD coordinate, the object is considered BAD and any computed TR/TA results will effectively be ignored in the final action determination.

Next, the boundary subset TR/TA results are combined to determine an overall status of the object. If the object is TR against any viewport or enabled UC plane, the object is considered TR. Note that, by definition, being TR against a VPXY boundary implies that the vertices will be TR agains the corresponding GB boundary, so computing TR_GB is unnecessary.

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way, objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

Finally, the object is considered MC if it is neither TR or TA.

The following logic is used to compute the final TR, TA, and MC status.

```
   —      //
   —      // ClipDetermination
   —      //
   —
   —      //
   —      // Compute objANDCode and objORCode
   —      //
   —      switch (object type) {
   —      case POINT:
   —      {
      —   objANDCode[…] = v0.outCode[…]
      —   objORCode[…]  = v0.outCode[…]
   —      } break
   —      case LINE:
   —      {
   —        objANDCode[…] = v0.outCode[…] & v1.outCode[…]
   —        objORCode[…]  = v0.outCode[…] | v1.outCode[…]
   —      } break
   —      case TRIANGLE:
   —      {
   —        objANDCode[…] = v0.outCode[…] & v1.outCode[…] &
v2.outCode[…]
   —        objORCode[…]  = v0.outCode[…] | v1.outCode[…] |
v2.outCode[…]
   —      } break
   —
   —      //
   —      // Determine TR/TA against interesting boundary subsets
   —      //
   —      TR_VPXY = (objANDCode[VP_L] | objANDCode[VP_R] |
objANDCode[VP_T] | objANDCode[VP_B])
   —      TR_GB   = (objANDCode[GB_L] | objANDCode[GB_R] |
objANDCode[GB_T] | objANDCode[GB_B])
   —      TA_GB   = !(objORCode[GB_L] | objORCode[GB_R] |
objORCode[GB_T] | objORCode[GB_B])
   —      TA_VPZ  = !(objORCode[VP_N] | objORCode[VP_Z])
   —      TR_VPZ  = (objANDCode[VP_N] | objANDCode[VP_Z])
   —      TA_UC   = !(objORCode[UC0] | objORCode[UC1] | … |
objORCode[UC7])
   —      TR_UC   = (objANDCode[UC0] | objANDCode[UC1] | … |
objANDCode[UC7])
   —      BAD     = objORCode[BAD]
   —
   —      //
   —      // Trivial Reject
   —      //
   —      //    An object is considered TR if all vertices are TR
   —      //    against any common boundary
   —      //    Note that this allows the case of the VPXY being
outside the GB
```

```
—       //
—
—       //
—       // Trivial Accept
—       //
—       // For an object to be TA, it must be TA against the
—       // VPZ and GB, not TR,
—       //   and considered TA against the UC planes
—       // If the UCMC mode is disabled, an object is
—       // considered TA against the UC
—       //   as long as it isn't TR against the UC.
—       // If the UCMC mode is enabled, then the object really
—       // has to be TA against the UC to be considered TA
—       //   In this way, enabling the UCMC mode will force
—       //  clipping if the object is neither
—       //   TA or TR against the UC
—       //
—       #ifdef
—       TA = !TR && TA_GB && TA_VPZ
—       #endif
—       UCMC = CLIP_STATE.UserClipFlagsMustClipEnable
—       TA = TA && ( (UCMC && TA_UC) || (!UCMC && !TR_UC) )
—
—       //
—       // MustClip
—       // This is simply defined as not TA or TR
—       // Note that exactly one of TA, TR and MC will be set
—       //
—       MC = !(TA || TR)
—
```

6.5.3 ClipMode

The ClipMode state determines what action the CLIP unit takes given the results of ClipDetermination. The possible actions are:

- PASSTHRU: Pass the object directly down the pipeline. A CLIP thread is not spawned.
- DISCARD: Remove the object from the pipeline and dereference object vertices as required (i.e., dereferencing will not occur if the vertices are shared with other objects).
- SPAWN: Pass the object to a CLIP thread. In the process of initiating the thread, the object vertices may be dereferenced.

The following logic is used to determine what to do with the object (PASSTHRU or DISCARD or SPAWN).

DevBW-E0, DevCL-B Errata: SPAWN is forced if the object is BAD and ClipMode is not REJECT_ALL

```
—      //
—      // Use the ClipMode to determine the action to take
—         //
—      switch (CLIP_STATE.ClipMode) {
—        case NORMAL: {
—         PASSTHRU = TA && !BAD
—         DISCARD  = TR || BAD
—         SPAWN    = MC && !BAD
—        }
—        case CLIP_ALL: {
—         PASSTHRU = 0
—         DISCARD  = 0
—         SPAWN    = 1
—        }
—        case CLIP_NOT_REJECT: {
—         PASSTHRU = 0
—         DISCARD  = TR || BAD
—         SPAWN    = !(TR || BAD)
—        }
—        case REJECT_ALL: {
—         PASSTHRU = 0
—         DISCARD  = 1
—         SPAWN    = 0
—        }
—        case ACCEPT_ALL: {
—         PASSTHRU = !BAD
—         DISCARD  = BAD
—         SPAWN    = 0
—        }
—      } endswitch
—
```

```
—       #ifdef (DevBW-E0 || DevCL-B)
—       if (BAD && CLIP_STATE.ClipMode != REJECT_ALL) {
—           DISCARD = 0
—           SPAWN = 1
—       }
—       #endif
```

6.5.3.1 NORMAL ClipMode

In NORMAL mode, objects will be discarded if TR or BAD, passed through if TA, and passed to a CLIP thread if MC. This mode is typically used when the CLIP kernel is only used to perform 3D Clipping (the expected usage model).

6.5.3.2 CLIP_ALL ClipMode

In CLIP_ALL mode, all objects (regardless of classification) will be passed to CLIP threads. Note that this includes BAD objects. This mode can be used to perform arbitrary processing in the CLIP thread, or as a backup if for some reason the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are not sufficient for controlling 3D Clipping.

6.5.3.3 CLIP_NON_REJECT ClipMode

This mode is similar to CLIP_ALL mode, but TR and BAD objects are discarded an all other (TA, MC) objects are passed to CLIP threads. Usage of this mode assumes that the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are sufficient at least in respect to determining trivial reject.

6.5.3.4 REJECT_ALL ClipMode

In REJECT_ALL mode, all objects (regardless of classification) are discarded. This mode effectively clips out all objects.

6.5.3.5 ACCEPT_ALL ClipMode

In ACCEPT_ALL mode, all non-BAD objects are passed directly down the pipeline. This mode partially disables the CLIP stage. BAD objects will still be discarded, and incomplete primitives (generated by a GS thread) will be discarded.

6.6 Object Pass-Through

Depending on ClipMode, objects may be passed directly down the pipeline. The PrimTopologyType associated with the output objects may differ from the input PrimTopologyType, as shown in the table below.

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
POINTLIST	POINTLIST	
POINTLIST_BF	POINTLIST_BF	
LINELIST	LINELIST	
LINESTRIP	LINESTRIP	
LINESTRIP_BF	LINESTRIP_BF	
LINESTRIP_CONT	LINESTRIP_CONT	
LINESTRIP_CONT_BF	LINESTRIP_CONT_BF	
LINELOOP	N/A	**Not supported after GS.**
TRILIST	TRILIST	
RECTLIST	RECTLIST	
TRISTRIP	TRISTRIP or TRISTRIP_REV	**Depends on where the incoming strip is broken (if at all) by discarded or clipped objects** **See Tristrip Clipping Errata subsection.**
TRISTRIP_REV	TRISTRIP or TRISTRIP_REV	**Depends on where the incoming strip is broken (if at all) by discarded or clipped objects** **See Tristrip Clipping Errata subsection.**
TRIFAN	TRIFAN	
TRIFAN_NOSTIPPLE	TRIFAN_NOSTIPPLE	
POLYGON	POLYGON	
QUADLIST	N/A	**Not supported after GS.**
QUADSTRIP	N/A	**Not supported after GS.**

6.7 CLIP Thread Request Generation

6.7.1 Object Vertex Ordering

The following table defines the number and order of object vertices passed in the Vertex Data portion of the CLIP thread payload, assuming an input topology with *N* vertices. The ObjectType passed to the thread is, by default, the incoming PrimTopologyType. Exceptions to this rule (for the TRISTRIP variants) are called out.

PrimTopologyType	Order of Vertices in Payload	Notes
<PRIMITIVE_TOPOLOGY>	[<object#>] = (<vert#>,…);	
POINTLIST	[0] = (0); [1] = (1); …; [N-2] = (N-2);	
POINTLIST_BF	Same as POINTLIST	**Handled same as POINTLIST**
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); …; [(N/2)-1] = (N-2,N-1)	
LINESTRIP (N >= 2)	[0] = (0,1); [1] = (1,2); …; [N-2] = (N-2,N-1)	
LINESTRIP_BF	Same as LINESTRIP	**Handled same as LINESTRIP**
LINESTRIP_CONT	Same as LINESTRIP	**Handled same as LINESTRIP**
LINESTRIP_CONT_BF	Same as LINESTRIP	**Handled same as LINESTRIP**
LINELOOP	N/A	**Not supported after GS.**
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); …; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST	Same as TRILIST	**Handled same as TRILIST**
TRISTRIP (N >= 3)	[0] = (0,1,2) {TRISTRIP} [1] = (1,2,3) {TRISTRIP_REVERSE}; … [N-3] = (N-3,N-2,N-1) {TRISTRIP or TRISTRIP_REVERSE}	**"Odd" triangles do not have vertices reordered, though identified as TRISTRIP_REVERSE so the thread knows this**

PrimTopologyType	Order of Vertices in Payload	Notes
TRISTRIP_REV (N >= 3)	[0] = (0,1,2) {TRISTRIP_REVERSE} [1] = (1,2,3) {TRISTRIP}; …; [N-3] = (N-3,N-2,N-1) {TRISTRIP or TRISTRIP_REVERSE}	**"Odd" triangles do not have vertices reordered, though identified as TRISTRIP so the thread knows this**
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); …; [N-3] = (0, N-2, N-1);	**Only used by OGL**
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON	Same as TRIFAN	
QUADLIST	N/A	**Not supported after GS.**
QUADSTRIP	**N/A**	**Not supported after GS.**

6.7.2 CLIP Thread Payload

Table 6-1 shows the layout of the payload delivered to CLIP threads.

Refer to 3D Pipeline Stage Overview (*3D Overview*) for details on those fields that are common amongst the various pipeline stages.

Table 6-1. CLIP Thread Payload

DWord	Bit	Description
R0.7	31	**Snapshot Flag.** If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. (See *Debug* for further description). Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the Scratch Space allocated to this thread, as an 1KB-aligned offset from the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:1	Reserved
	0	**FFTID.** This ID is assigned by the fixed function unit and is a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:4	Reserved
	3:0	**Per Thread Scratch Space.** Specifies the amount of Scratch Space allocated to this thread, as a power of 2 bytes in excess of 1KB. Format = U4 Range = [0,11] indicating [1KB, 2MB] in powers of two
R0.2	31	**Object Outcode [VP.XMin].** This bit contains the logical OR of the VP.XMin vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the VP.XMin clip plane). **Pre-DevBW-E0, DevCL-A Errata:** All VP and GB ObjectOutcodes are UNDEFINED if any vertex of the object has a negative RHW component. See W Clipping Errata sections above for more information.
	30	**Object Outcode [VP.XMax]**

DWord	Bit	Description
	29	**Object Outcode [VP.YMin]**
	28	**Object Outcode [VP.YMax]**
	27	**Object Outcode [VP.ZMin]**
	26	**Object Outcode [VP.ZMax]**
	25	**Object Outcode [GB.XMin].** This bit contains the logical OR of the GB.XMin vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the GB.XMin clip plane).
	24	**Object Outcode [GB.XMax]**
	23	**Object Outcode [GB.YMin]**
	22	**Object Outcode [GB.YMax]**
	21	**Object Outcode [UserClip7].** This bit contains the logical OR of the UserClip7 vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the UserClip7 clip plane). **Pre-DevBW-E0,DevCL-B Errata:** This bit is the logical OR of the NEGW outcodes over all the vertices of the object.
	20	**Object Outcode [UserClip6]**
	19	**Object Outcode [UserClip5]**
	18	**Object Outcode [UserClip4]**
	17	**Object Outcode [UserClip3]**
	16	**Object Outcode [UserClip2]**
	15	**Object Outcode [UserClip1]**
	14	**Object Outcode [UserClip0]**
	13	**Object Outcode [BAD]** **Note:** If set, all VP and GB-related Object Outcodes are UNDEFINED.
	12	Reserved
	11:10	Reserved
	9	**Edge Indicator [1].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex2 to Vertex0 is an exterior edge of the polygon (i.e., this is the last or only triangle of the polygon). If clear, that edge is an interior edge. The CLIP kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved 0 = V2→V0 is not an outside edge 1 = V2→V0 is an outside edge

DWord	Bit	Description
	8	**Edge Indicator [0].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex0 to Vertex1 is an exterior edge of the polygon (i.e., this is the first or only triangle of the polygon). If clear, that edge is an interior edge. The CLIP kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved 0 = V0→V1 is not an outside edge 1 = V0→V1 is an outside edge
	7:5	Reserved
	4:0	**Primitive Topology Type.** This field identifies the "basic" Primitive Topology Type associated with the primitive spawning this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the CLIP unit may toggle this value between TRISTRIP and TRISTRIP_REV, as described in 6.7.1. Format: (See 3DPRIMITIVE command in *3D Pipeline*)
R0.1	31:0	Reserved
R0.0	31:23	Reserved
	22:16	**Handle ID.** This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. Format: Reserved for Implementation Use
	15:9	Reserved
	8:0	**URB Return Handle.** This is the initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the first destination URB entry. Format: U9 URB Handle
[Varies] optional	31:0	**Constant Data** (optional). Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state.
Varies	31:0	**Vertex Data.** There can be up to 3 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state Vertex 0 DWord 0 is located at Rn.0, Vertex 0 DWord 1 is located at Rn.1, etc. Vertex 1 DWord 0 immediately follows the last DWord of Vertex 0, and so on. See Object Vertex Ordering (above) for a definition of the number and order of vertices passed in the payload.

6.8 CLIP Thread Execution

A CLIP kernel can perform arbitrary operations on the input object. Input data is either passed directly in the thread payload (including the input object vertex data) or indirectly via pointers passed in the payload. It is anticipated that the CLIP kernel

implement a 3D clipping algorithm though this is not strictly required. Definition of candidate algorithms is beyond the scope of this document.

Refer to *Jitter Requirements* for any other system requirements on the CLIP kernel. Refer to 3D Pipeline Stage Overview (*3D Overview*) for further information on FF-unit/Thread interactions.

6.8.1 Vertex Output

The CLIP thread can output a number (possibly zero) of destination VUEs. Refer to Thread Output Handling (*3D Overview*).

A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a "retained" handle refers to a URB handle that (a) has been pre-allocated or allocated and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

6.8.2 Thread Termination

CLIP threads must signal thread termination by issuing a URB_WRITE message to the URB shared function with the **EOT** and **Complete** bits set.

6.9 Thread-Generated Vertex Readback

The CLIP unit performs a readback of the Vertex Header of each vertex output by a CLIP thread, as this information is required by the next stage (SF). Note that trivially-accepted vertices (not generated by a CLIP thread) already have been readback in the GS stage. See *Vertex Data Overview* for a description of the Vertex Header fields and how they are read-back and used by the CLIP unit.

The CLIP unit will extract the following per-vertex readback data and associate it with the generated vertex as it is sent down the pipeline:

- PrimTopologyType
- PrimStart
- PrimEnd
- Viewport Index
- RenderTarget Array Index
- PointWidth
- Vertex Position X,Y,Z,RHW (NDC coordinates only)

Note that the UserClipFlags are not read back as they are not relevant past the clip stage.

6.10 Primitive Output

(This section refers to output from the CLIP unit to the pipeline, not output from the CLIP thread)

The CLIP unit will output primitives (either passed-through or generated by a CLIP thread) in the proper order. This includes the buffering of a concurrent CLIP thread's output until the preceding CLIP thread terminates. Note that the requirement to buffer subsequent CLIP thread output until the preceding CLIP thread terminates has ramifications on determining the number of VUEs allocated to the CLIP unit and the number of concurrent CLIP threads allowed.

6.11 Other Functionality

6.11.1 Statistics Gathering

The CLIP unit includes logic to assist in the gathering of certain pipeline statistics, primarily in support of the Asynchronous Query function of the D3D APIs. The statistics take the form of MI counter registers (see *Memory Interface Registers*), where the CLIP unit provides signals causing those counters to increment.

Software is responsible for controlling (enabling) these counters in order to provide the required statistics at the DDI level. For example, software might need to disable the statistics gathering before submitting non-API-visible objects (e.g., RECTLISTs) for processing.

The CLIP unit must be ENABLED (via the **CLIP Enable** bit of PIPELINED_STATE_POINTERS) in order to it to affect the statistics counters. This might lead to a pathological case where the CLIP unit needs to be ENABLED simply to provide statistics gathering. If no clipping functionality is desired, **Clip Mode** can be set to ACCEPT_ALL to effectively inhibit clipping while leaving the CLIP stage ENABLED.

The two statistics the CLIP unit affects (if enabled) are:

- CL_INVOCATION_COUNT: Incremented for every CLIP thread spawned.
- GS_PRIMITIVES_COUNT: Incremented for every object received from the GS stage.

Implementation Note: The reason the CLIP unit counts GS-produced objects (and, similarly, why the SF unit counts CLIP-produced objects) is that a downstream Object Staging Buffer is an opportunistic place to count objects generated by threads in an upstream unit.

6.11.1.1 CL_INVOCATION_COUNT

If the **Statistics Enable** bit (CLIP_STATE) is set, the CLIP unit increments the CL_INVOCATION_COUNT register each time a CLIP thread is spawned.

6.11.1.2 GS_PRIMITIVES_COUNT

If **GS Output Object Statistic Enable** is set (CLIP_STATE), the CLIP stage increments GS_PRIMITIVES_COUNT for every complete object received from the GS stage.

In order to maintain a count of objects generated by the API's Geometry Shader function (presumably the number of objects output by GS threads), software will need to clear the CLIP unit's **GS Output Object Statistic Enable** whenever the GS unit is DISABLED.

§§

7 Strips and Fans (SF) Stage

7.1 Overview

The Strips and Fan (SF) stage of the GEN4 3D pipeline is responsible for performing "setup" operations required to rasterize 3D objects.

This functionality is split between fixed-function hardware in the SF unit and SF (aka Setup) threads spawned to compute plane equations required for attribute interpolation.

7.1.1 Inputs from CLIP

The following table describes the per-vertex inputs passed to the SF unit from the previous (CLIP) stage of the pipeline.

Table 7-1. SF's Vertex Pipeline Inputs

Variable	Type	Description
primType	enum	Type of primitive topology the vertex belongs to. See Table 7-2 for a list of primitive types supported by the SF unit. See *3D Pipeline* for descriptions of these topologies. QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit. Software must use a GS thread to convert these to some other (supported) primitive type.
primStart,primEnd	boolean	Indicate vertex's position within the primitive topology
vInX[]	float	Vertex X position (screen space or NDC space)
vInY[]	float	Vertex Y position (screen space or NDC space)
vInZ[]	float	Vertex Z position (screen space or NDC space)
vInInvW[]	float	Reciprocal of Vertex homogeneous (clip space) W
hVUE[]	URB address	Points to the vertex's data stored in the URB (one VUE handle per vertex)
renderTargetArrayIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in subsequent processing.

Variable	Type	Description
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in the Viewport Transform and Scissor subfunctions, otherwise the value is ignored. Note that for primitive topologies with vertices shared between objects, this means a shared vertex may be subject to multiple Viewport Transformation operations if the viewPortIndex varies within the topology.
pointSize	uint	If this vertex is within a POINTLIST[_BF] primitive topology, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise the value is ignored.

7.1.2 Attribute Setup / Interpolation Process

Required inputs to API Pixel Shader programs are the pixel's position and interpolated vertex attributes sampled at the pixel position. In order to produce these per-pixel parameters, certain setup calculations need to be performed within the SF stage to provide inputs for the subsequent interpolation process at in the WM stage. Where and how the setup calculations and interpolation are performed varies by attribute (due to various cost/performance tradeoffs), as outlined below:

- **Position X,Y**: The SF unit performs the position X,Y setup computations in fixed function hardware and passes these results directly to the WM stage. The WM unit interpolates position (i.e., rasterizes the object) in fixed-function hardware and passes pixel X,Y information to the WM (PS) thread in the thread's payload.
- **Position Z**: The handling of the position Z attribute is more complicated. The SF unit performs operations to compute Z at object vertices (e.g., viewport map, etc.). The object vertex's position Z values are then passed to the SF (Setup) thread in the fixed header portion of the thread payload. The SF thread is responsible for performing the setup computations for position Z and storing the required result values (plane equation coefficients) in the PUE. Subsequently the WM unit will directly read the position Z plane equation coefficients from the PUE (at a location programmed via WM_STATE). The WM unit will then perform interpolation of position Z (along with some other computations like Depth Offset, etc.) to derive per-pixel position Z. This value is then used for Early Depth Test, if applicable. The per-pixel position Z value ("source depth") can be optionally included in WM thread payloads for use by the thread.
- **Position 1/W**: This attribute could be handled like "other vertex attributes", but as an optimization it is treated slightly differently. The position 1/W values of the object vertices are passed from the SF unit to the SF thread in the fixed header portion of the thread payload. These values are unmodified copies of the postion 1/W values read back from the object's VUEs – so in theory the SF thread could use the values from the VUEs like it does for other attributes. However, having the SF unit insert them into the payload allows software to avoid having the 256-bit Vertex Headers read from the VUEs and placed in the SF thread payload, thus removing this traffic from the thread dispatch process. The SF thread performs setup computations on the position 1/W attributes and stores the results in the output PUE. Subsequently, the WM thread will use the position 1/W setup parameters to interpolate position 1/W to the pixel location. This is likely one of

the first operations in the PS thread, as the pixel's interpolated 1/W value is required to perform perspective-correct interpolation of other vertex attributes.

- **Other Vertex Attributes**: The handling of non-position vertex attributes (e.g., texture coordinates, colors, etc.) is straightforward. The SF unit is not directly involved with the setup computations for these attributes, and the WM unit is not directly involved with their interpolation. The SF thread will use the object's vertex attributes provided in the VUE data in the thread payload, perform the setup computations as required, and store the results in the output PUE. The WM thread will use this PUE data to interpolate the attributes to the pixel location.

7.1.3 Outputs to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects. The types of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- Handle of the Primitive URB Entry (PUE) that was written by the SF (Setup) thread. This handle will be passed to all WM (PS) threads spawned from the WM's rasterization process.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.)
- Edge or line interpolation information (e.g., edge equation coefficients, etc.)
- Information on where the WM is to start rasterization of the object
- Object orientation (front/back-facing)
- Last Pixel indication (for line drawing)

7.2 Primitive Assembly

The first subfunction within the SF unit is *Primitive Assembly*. Here 3D primitive vertex information is buffered and, when a sufficient number of vertices are received, converted into basic 3D objects which are then passed to the Viewport Transformation subfunction.

The number of vertices passed with each primitive is constrained by the primitive type and must conform to Table 7-2. Passing any other number of vertices results in UNDEFINED behavior. Note that this restriction only applies to primitive output by GS threads (which is under control of the GS kernel). See the Vertex Fetch chapter for details on how the VF unit automatically removes incomplete objects resulting from processing a 3DPRIMITIVE command.

Table 7-2. SF-Supported Primitive Types & Vertex Count Restrictions

primType	VertexCount Restriction
3DPRIM_TRILIST	nonzero multiple of 3
3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE	>=3
3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	>=3
3DPRIM_LINELIST	nonzero multiple of 2
3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	>=2
3DPRIM_RECTLIST	nonzero multiple of 3
3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	nonzero

The 3D object types are listed in Table 7-3.

Table 7-3. 3D Object Types

objectType	generated by *primType*	Vertices/Object
3DOBJ_POINT	3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	1
3DOBJ_LINE	3DPRIM_LINELIST 3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	2
3DOBJ_TRIANGLE	3DPRIM_TRILIST 3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE 3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	3
3DOBJ_RECTANGLE	3DPRIM_RECTLIST	3 (expanded to 4 in RectangleCompletion)

The outputs of Primitive Decomposition are listed in Table 7-4.

Table 7-4. Primitive Decomposition Outputs

Variable	Type	Description
objectType	enum	Type of object. See Table 7-3
nV	uint	The number of object vertices passed to Object Setup. See Table 7-3
v[0..nV-1]	various	Data arrays associated with object vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. See Table 7-3
invertOrientation	enum	Indicates whether the orientation (CW or CCW winding order) of the vertices of a triangle object should be inverted. Ignored for non-triangle objects.
backFacing	enum	Valid only for points and line objects, indicates a back facing object. This is used later for culling.
provokingVtx	uint	Specifies the index (into the *v[]* arrays) of the vertex considered the "provoking" vertex (for flat shading). The selection of the provoking vertex is programmable via SF_STATE (**xxx Provoking Vertex Select** state variables.)
polyStippleEnable	boolean	TRUE if Polygon Stippling is enabled. FALSE for TRIFAN_NOSTIPPLE. Ignored for non-triangle objects.
continueStipple	boolean	Only applies to line objects. TRUE if Line Stippling should be continued (i.e., not reset) from where the previous line left off. If FALSE, Line Stippling is reset for each line object.
renderTargetIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. This value is simply passed in SF thread payloads and not used within the SF unit.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object.
pointSize	unit	For point objects, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise the value is ignored.

The following table defines, for each primitive topology type, which vertex's VPIndex/RTAIndex applies to the objects within the topology.

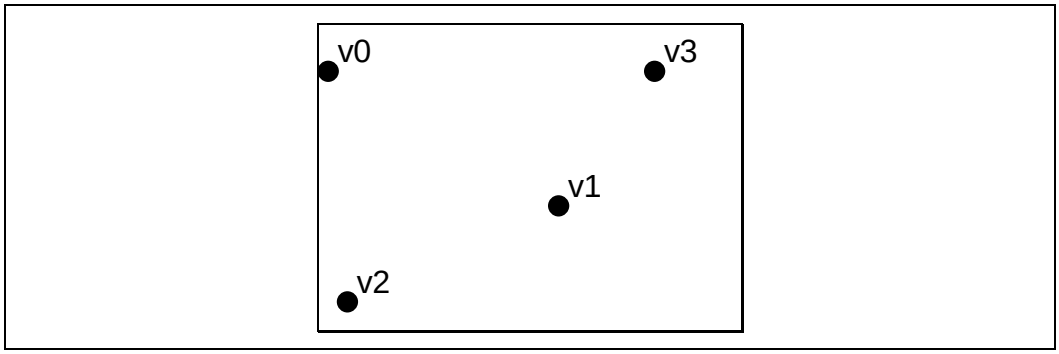
Table 7-5. VPIndex/RTAIndex Selection

PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V2.VPIndex → Line(V2,V3) …
LINESTRIP LINESTRIP_BF LINESTRIP_CONT LINESTRIP_CONT_BF	The leading vertex of each line segment supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V1.VPIndex → Line(V1,V2) … NOTE: If the VPIndex changes within the topology, shared vertices will be processed (mapped) multiple times.
TRILIST RECTLIST	The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects. V0.VPIndex → Tri(V0,V1,V2) V3.VPIndex → Tri(V3,V4,V5) …
TRISTRIP TRISTRIP_REVERSE	The leading vertex of each triangle supplies the VPIndex for the corresponding triangle object. V0.VPIndex → Tri(V0,V1,V2) V1.VPIndex → Tri(V1,V2,V3) … NOTE: If the VPIndex changes within the primitive, shared vertices will be processed (mapped) multiple times.
TRIFAN TRIFAN_NOSTIPPLE POLYGON	The first vertex (V0) supplies the VPIndex for all triangle objects.

7.2.1 Point List Decomposition

The 3DPRIM_POINTLIST and 3DPRIM_POINTLIST_BACKFACING primitives specify a list of independent points.

Figure 7-1. 3DPRIM_POINTLIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_POINT objects that are then passed individually and in order to the Object Setup subfunction. The *provokingVertex* of each object is, by definition, v[0].

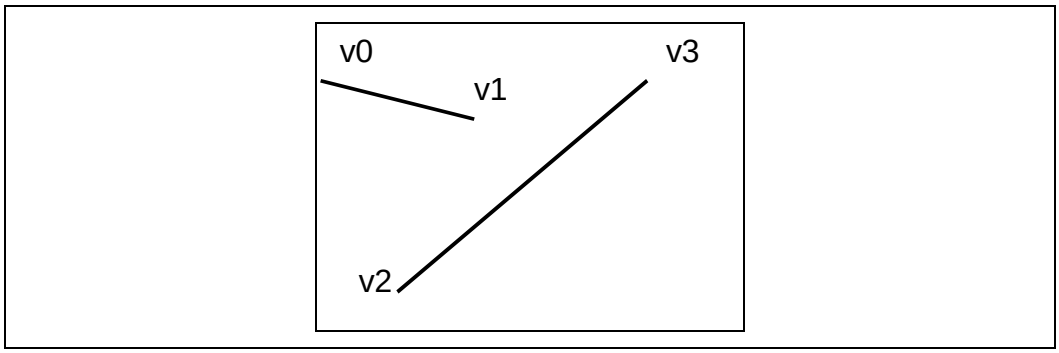
Points have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing points. Primitives of type 3DPRIM_POINTLIST_BACKFACING are decomposed exactly the same way as 3DPRIM_POINTLIST primitives, but the *backFacing* variable is set for resulting point objects being passed on to object setup.

```
PointListDecomposition() {
   objectType = 3DOBJ_POINT
   nV = 1
      provokingVtx = 0
   if (primType == 3DPRIM_POINTLIST)
            backFacing = FALSE
   else  // primType == 3DPRIM_POINTLIST_BACKFACING
            backFacing = TRUE
      for each (vertex I in [0..vertexCount-1]) {
      v[0] ← vIn[i]      // copy all arrays (e.g., v[]X, v[]Y, etc.)
      ObjectSetup()
   }
}
```

7.2.2 Line List Decomposition

The 3DPRIM_LINELIST primitive specifies a list of independent lines.

Figure 7-2. 3DPRIM_LINELIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0, v1; v2, v3; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

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

7.2.3 Line Strip Decomposition

The 3DPRIM_LINESTRIP, 3DPRIM_LINESTRIP_CONT, 3DPRIM_LINESTRIP_BF, and 3DPRIM_LINESTRIP_CONT_BF primitives specify a list of connected lines.

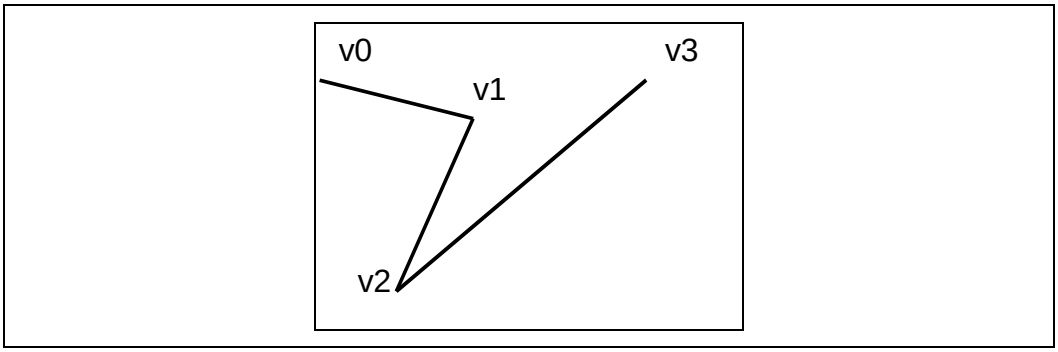


Figure 7-3. 3DPRIM_LINESTRIP_xxx Primitive

The decomposition process divides the strip into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0,v1; v1,v2; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

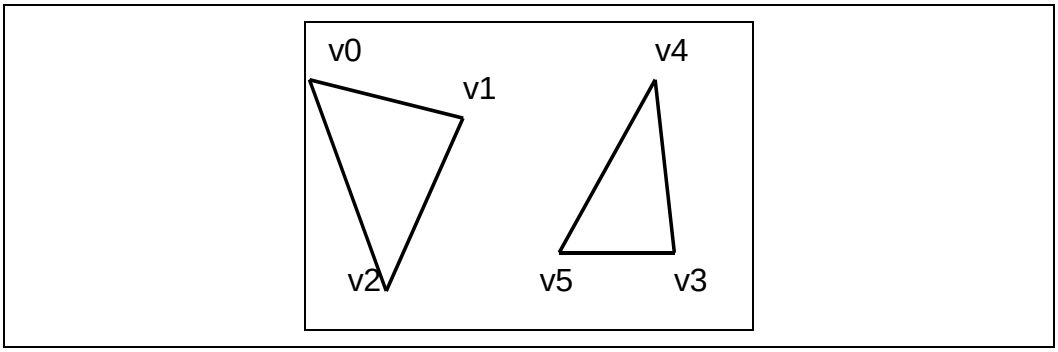
Lines have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing lines. Primitives of type 3DPRIM_LINESTRIP[_CONT]_BF are decomposed exactly the same way as 3DPRIM_LINESTRIP[_CONT] primitives, but the *backFacing* variable is set for the resulting line objects being passed on to object setup. Likewise 3DPRIM_LINESTRIP_CONT[_BF] primitives are decomposed identically to basic line strips, but the *continueStipple* variable is set to true so that the line stipple pattern will pick up from where it left off with the last line primitive, rather than being reset.

```
LineStripDecomposition() {
   objectType = 3DOBJ_LINE
   nV = 2
      provokingVtx = Line List/Strip Provoking Vertex Select
   if (primType == 3DPRIM_LINESTRIP) {
            backFacing = FALSE
            continueStipple = FALSE
   } else if (primType == 3DPRIM_LINESTRIP_BF) {
            backFacing = TRUE
            continueStipple = FALSE
   } else if (primType == 3DPRIM_LINESTRIP_CONT) {
            backFacing = FALSE
            continueStipple = TRUE
   } else if (primType == 3DPRIM_LINESTRIP_CONT_BF) {
            backFacing = TRUE
            continueStipple = TRUE
   }
      for each (vertex I in [0..vertexCount-1]) {
      v[0] arrays ← vIn[i] arrays
      v[1] arrays ← vIn[i+1] arrays
      ObjectSetup()
      continueStipple = TRUE
   }
}
```

7.2.4 Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

Figure 7-4. 3DPRIM_TRILIST Primitive



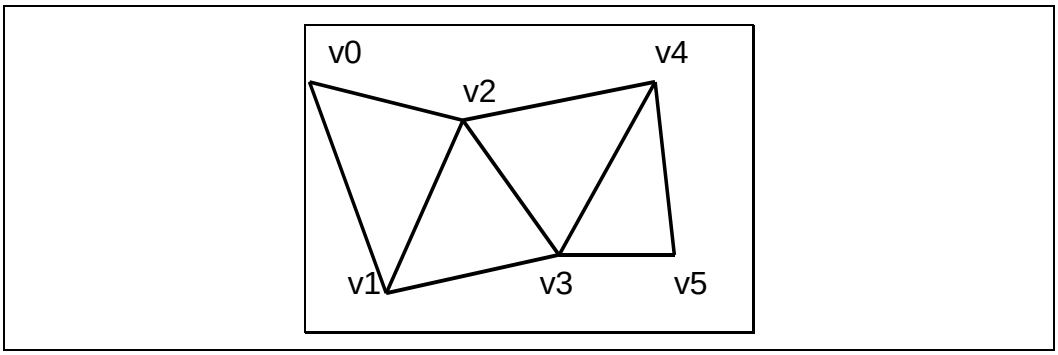
The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

```
TriangleListDecomposition() {
   objectType = 3DOBJ_TRIANGLE
   nV = 3
      invertOrientation = FALSE
   provokingVtx = Triangle List/Strip Provoking Vertex Select
      polyStippleEnable = TRUE
      for each (vertex I in [0..vertexCount-3] by 3) {
      v[0] arrays ← vIn[i] arrays
      v[1] arrays ← vIn[i+1] arrays
      v[2] arrays ← vIn[i+2] arrays
      ObjectSetup()
   }
}
```

7.2.5 Triangle Strip Decomposition

The 3DPRIM_TRISTRIP and 3DPRIM_TRISTRIP_REVERSE primitives specify a series of triangles arranged in a strip, as illustrated below.

Figure 7-5. 3DPRIM_TRISTRIP[_REVERSE] Primitive



The decomposition process divides the strip into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v1,v2,v3; v2,v3,v4; and so on. Note that the *winding order* of the vertices alternates between CW (clockwise), CCW (counter-clockwise), CW, etc. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

The 3D pipeline uses the winding order of the vertices to distinguish between front-facing and back-facing triangles (see Triangle Orientation (Face) Culling below). Therefore, the 3D pipeline must account for the alternation of winding order in strip triangles. The *invertOrientation* variable is generated and used for this purpose.

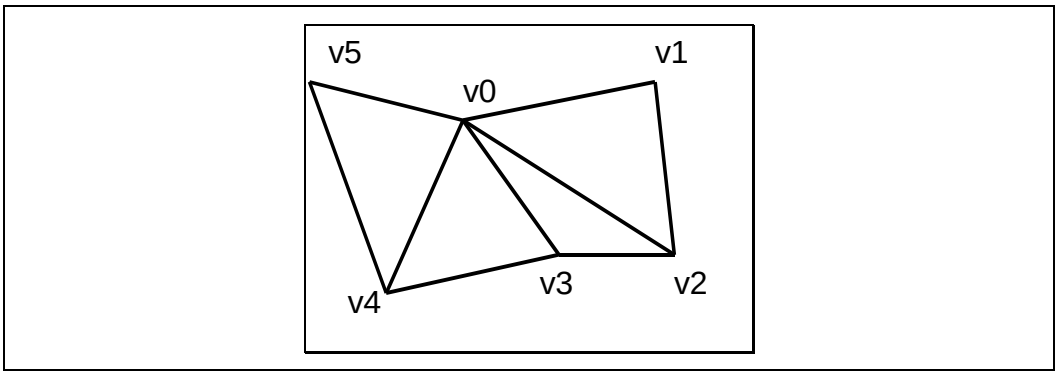
To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (e.g., due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

```
TriangleStripDecomposition() {
    objectType = 3DOBJ_TRIANGLE
    nV = 3
        provokingVtx = Triangle List/Strip Provoking Vertex Select
    if (primType == 3DPRIM_TRISTRIP)
            invertOrientation = FALSE
    else  // primType == 3DPRIM_TRISTRIP_REVERSE
            invertOrientation = TRUE
    polyStippleEnable = TRUE
        for each (vertex I in [0..vertexCount-3]) {
        v[0] arrays ← vIn[i] arrays
        v[1] arrays ← vIn[i+1] arrays
        v[2] arrays ← vIn[i+2] arrays
        ObjectSetup()
        invertOrientation = ! invertOrientation
    }
}
```

7.2.6 Triangle Fan Decomposition

The 3DPRIM_TRIFAN and 3DPRIM_TRIFAN_NOSTIPPLE primitives specify a series of triangles arranged in a fan, as illustrated below.

Figure 7-6. 3DPRIM_TRIFAN Primitive



The decomposition process divides the fan into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v0,v2,v3; v0,v3,v4; and so on. As there is no alternation in the vertex winding order, the *invertOrientation* variable is output as FALSE unconditionally. The *provokingVertex* of each object is taken from the **Triangle Fan Provoking Vertex** state variable, as programmed via SF_STATE.

Primitives of type 3DPRIM_TRIFAN_NOSTIPPLE are decomposed exactly the same way, except the *polyStippleEnable* variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```
TriangleFanDecomposition() {
       objectType = 3DOBJ_TRIANGLE
   nV = 3
       invertOrientation = FALSE
   provokingVtx = Triangle Fan Provoking Vertex Select
   if (primType == 3DPRIM_TRIFAN)
            polyStippleEnable = TRUE
   else  // primType == 3DPRIM_TRIFAN_NOSTIPPLE
            polyStippleEnable = FALSE
   v[0] arrays ← vIn[0] arrays  // the 1st vertex is common
      for each (vertex I in [1..vertexCount-2]) {
      v[1] arrays ← vIn[i] arrays
      v[2] arrays ← vIn[i+1] arrays
      ObjectSetup()
   }
}
```

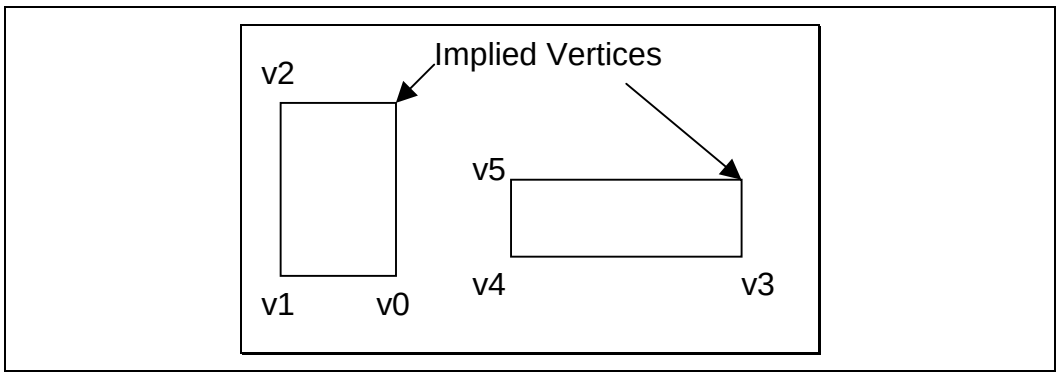
7.2.7 Polygon Decomposition

The 3DPRIM_POLYGON primitive is identical to the 3DPRIM_TRIFAN primitive with the exception that the *provokingVtx* is overridden with 0. This support has been added specifically for OpenGL support, avoiding the need for the driver to change the provoking vertex selection when switching between trifan and polygon primitives.

7.2.8 Rectangle List Decomposition

The 3DPRIM_RECTLIST primitive command specifies a list of independent, axis-aligned rectangles. Only the lower right, lower left, and upper left vertices (in that order) are included in the command – the upper right vertex is derived from the other vertices (in Object Setup).

Figure 7-7. 3DPRIM_RECTLIST Primitive



The decomposition of the 3DPRIM_RECTLIST primitive is identical to the 3DPRIM_TRILIST decomposition, with the exception of the *objectType* variable.

```
RectangleListDecomposition() {
   objectType = 3DOBJ_RECTANGLE
   nV = 3
      invertOrientation = FALSE
   provokingVtx = 0
      for each (vertex I in [0..vertexCount-3] by 3) {
      v[0] arrays ← vIn[i] arrays
      v[1] arrays ← vIn[i+1] arrays
      v[2] arrays ← vIn[i+2] arrays
      ObjectSetup()
   }
}
```

7.3 Object Setup

The Object Setup subfunction of the SF stage takes the post-viewport-transform data associated with each vertex of a basic object and computes various parameters required for scan conversion. This includes generation of implied vertices, translations and adjustments on vertex positions, and culling (removal) of certain classes of objects. The final object information is passed to the Windower/Masker (WM) stage where the object is rasterized into pixels.

7.3.1 Invalid Position Culling (Pre/Post-Transform)

At input the the SF stage, any objects containing a floating-point NaN value for Position X, Y, Z, or RHW will be unconditionally discarded. Note that this occurs on an object (not primitive) basis.

If Viewport Transformation is enabled, any objects containing a floating-point NaN value for post-transform Position X, Y or Z will be unconditionally discarded.

7.3.2 Viewport Transformation

If the **Viewport Transform Enable** bit of SF_STATE is ENABLED, a viewport transformation is applied to each vertex of the object.

The VPIndex associated with the leading vertex of the object is used to obtain the **Viewport Matrix Element** data from the corresponding element of the SF_VIEWPORT structure in memory. For each object vertex, the following scale and translate transformation is applied to the position coordinates:

x' = **m00** * x + **m30**

y' = **m11** * y + **m31**

z' = **m22** * z + **m32**

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

7.3.3 Destination Origin Bias

The positioning of the pixel sampling grid is programmable and is controlled by the **Destination Origin Horizontal/Vertical Bias** state variables (set via SF_STATE). If these bias values are both 0, pixels are sampled on an integer grid. Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0,0) falls within the primitive. This positioning of the sample grid corresponds with the rasterization rules where "pixel centers are on an integer grid".

If the bias values are both 0.5, pixels are sampled on a "half" integer grid (i.e., X.5, Y.5). Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0.5,0.5) falls within the primitive. This positioning of the sample grid corresponds with the OpenGL rasterization rules, where "fragment centers" lay on a half-integer grid. It also corresponds with the Intel740 rasterizer (though that device did not employ "top left" rules).

Note that subsequent descriptions of rasterization rules for the various objects will be with reference to the pixel sampling grid.

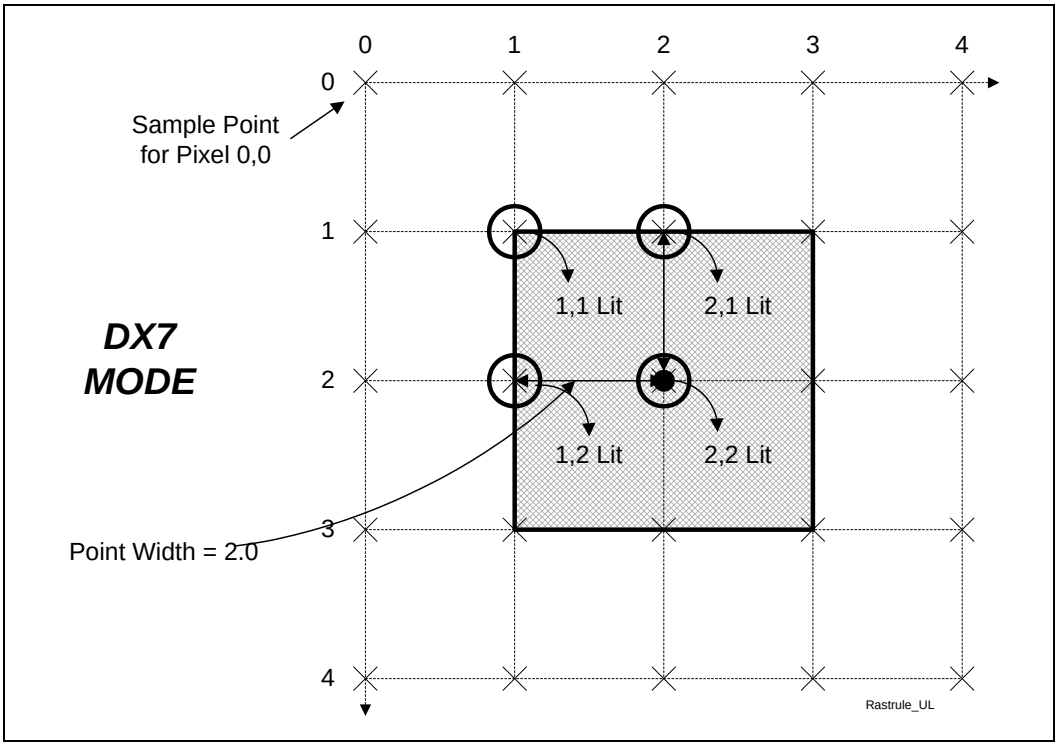


Figure 7-8. Destination Origin Bias

7.3.4 Point Rasterization Rule Adjustment

POINT objects are rasterized as square RECTANGLEs, with one exception: The **Point Rasterization Rule** state variable (in SF_STATE) controls the rendering of point object edges that fall directly on pixel sample points, as the treatment of these edge pixels varies between APIs.

The following diagram shows the rasterization of a 2-pixel wide point centered at (2,2) Here the pixel sample grid coincides with the integer pixel coordinates, and the **Point Rasterization Rule** is set to RASTRULE_UPPER_LEFT. Note that the pixels which lie only on the upper and/or left edges are lit.

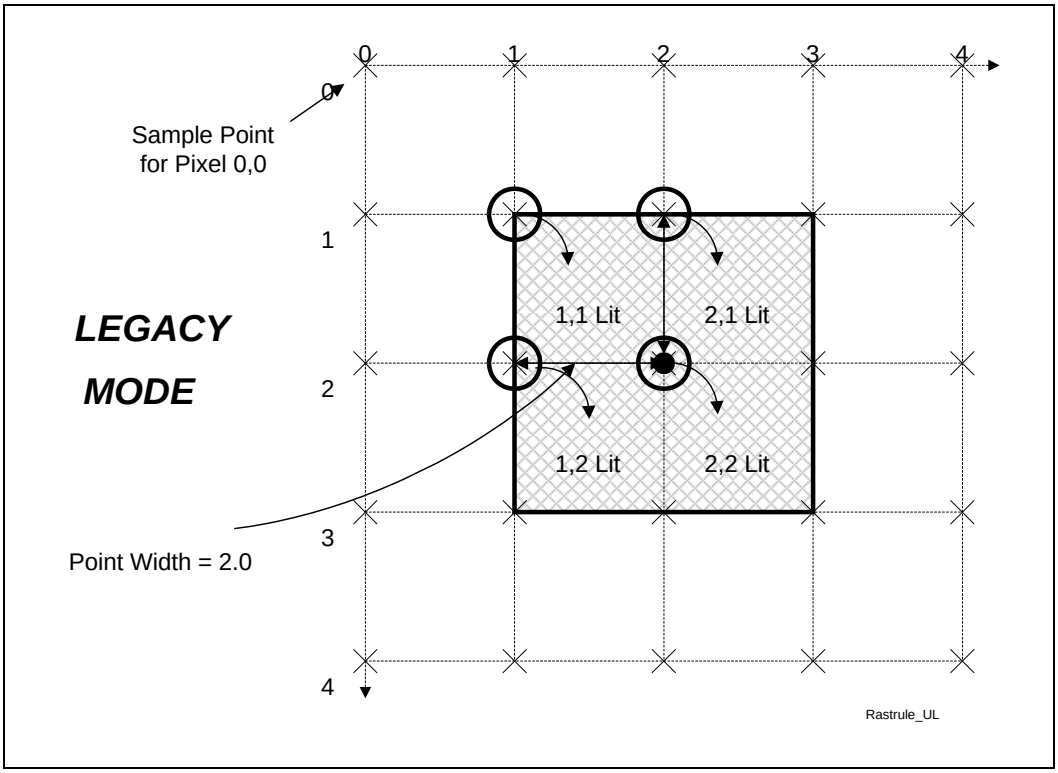
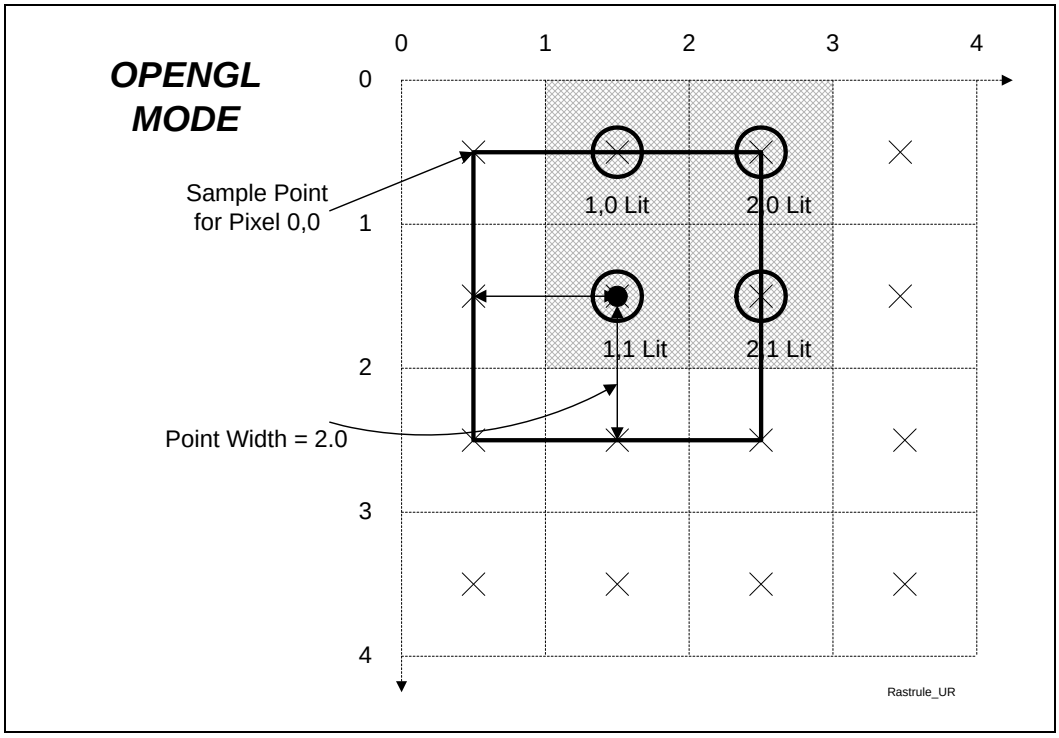


Figure 7-9. RASTRULE_UPPER_LEFT

The following diagram shows the rasterization of a 2-pixel wide point centered at (2,2) given “OpenGL” rasterization rules. Here the pixel sample grid coincides with half-integer pixel coordinates, and the **Point Rasterization Rule** is set to RASTRULE_UPPER_RIGHT. Note that the pixels which lie only on the upper and/or right edges are lit.

Figure 7-10. RASTRULE_UPPER_RIGHT



7.3.5 Drawing Rectangle Offset Application

The Drawing Rectangle Offset subfunction offsets the object’s vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window’s or viewport’s upper-left corner.

Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the “clipped” draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.

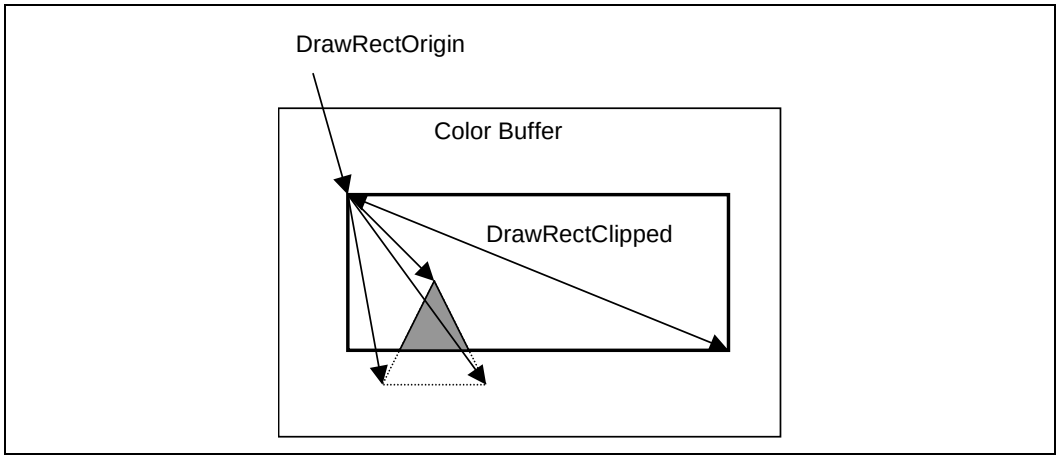


Figure 7-11. Onscreen Draw Rectangle

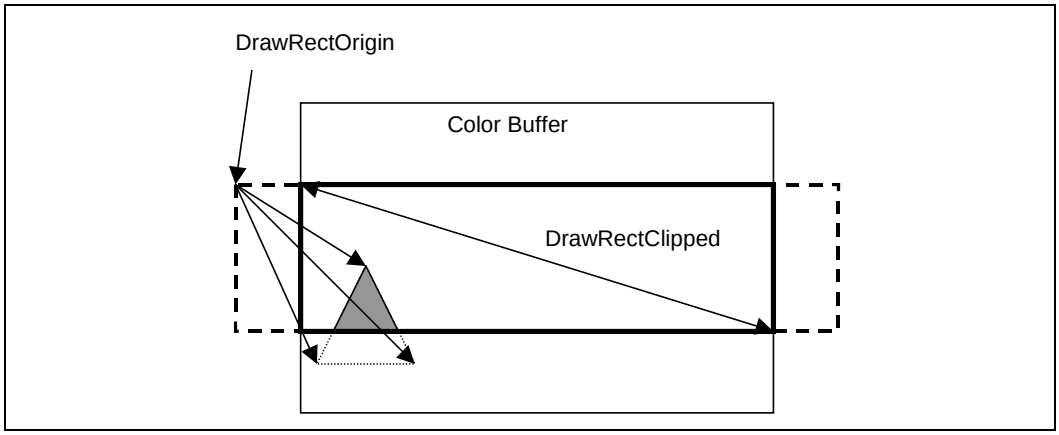


Figure 7-12. Partially-offscreen Draw Rectangle

7.3.5.1 3DSTATE_DRAWING_RECTANGLE

3DSTATE_DRAWING_RECTANGLE			
Project:	All	**Length Bias:**	2
The 3DSTATE_DRAWING_RECTANGLE command is used to set the 3D drawing rectangle and related state.			

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 00h 3DSTATE_DRAWING_RECTANGLE 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 Project: All
1	31:16	**Clipped Drawing Rectangle Y Min** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 31:29) Specifies Ymin value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with Y coordinates *less than* Ymin will be clipped out. **Programming Notes** This value must be *less than or equal to* **Clipped Drawing Rectangle Y Max.** If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.

3DSTATE_DRAWING_RECTANGLE		
	15:0	**Clipped Drawing Rectangle X Min** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 15:13) Specifies Xmin value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with X coordinates *less than* Xmin will be clipped out. **Programming Notes** **Project** This value must be *less than or equal to* **Clipped Drawing Rectangle X Max.** If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. All
2	31:16	**Clipped Drawing Rectangle Y Max** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 31:29) Specifies Ymax value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with coordinates *greater than* Ymax will be clipped out. **Programming Notes** This value must be *greater than or equal to* **Clipped Drawing Rectangle Y Min.** If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.
	15:0	**Clipped Drawing Rectangle X Max** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 15:13) Specifies Xmax value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with coordinates *greater than* Xmax will be clipped out. **Programming Notes** **Project** This value must be *greater than or equal to* **Clipped Drawing Rectangle X Min.** If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. All

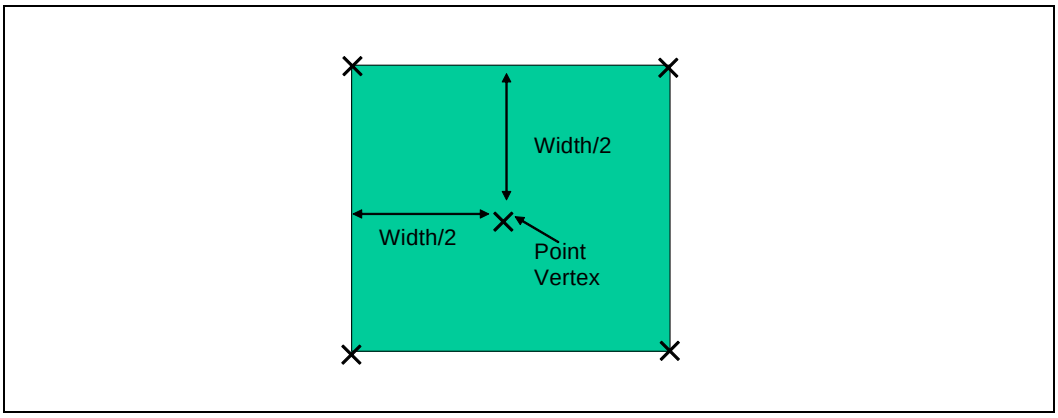
3DSTATE_DRAWING_RECTANGLE		
3	31:16	**Drawing Rectangle Origin Y** Project: All Format: S15 in Pixels from Color Buffer origin (upper left corner). FormatDesc Range [-8192,8191] (Bits 31:30 should be a sign extension) Specifies Y origin of Drawing Rectangle (in whole pixels) relative to origin of the Color Buffer, used to map incoming (Draw Rectangle-relative) vertex positions to the Color Buffer space.
	15:0	**Drawing Rectangle Origin X** Project: All Format: S15 in Pixels from Color Buffer origin (upper left corner). FormatDesc Range [-8192,8191] (Bits 15:14 should be a sign extension) Specifies X origin of Drawing Rectangle (in whole pixels) relative to origin of the Color Buffer, used to map incoming (Draw Rectangle-relative) vertex positions to the Color Buffer space.

7.3.6 Point Width Application

This stage of the pipeline applies only to 3DOBJ_POINT objects. Here the point object is converted from a single vertex to four vertices located at the corners of a square centered at the point's X,Y position. The width and height of the square are specified by a *point width* parameter. The **Use Point Width State** value in SF_STATE determines the source of the point width parameter: the point width is either taken from the **Point Width** value programmed in SF_STATE or the PointWidth specified with the vertex (as read back from the vertex VUE earlier in the pipeline).

The corner vertices are computed by adding and subtracting one half of the point width, as shown in Figure 7-13.

Figure 7-13. Point Width Application

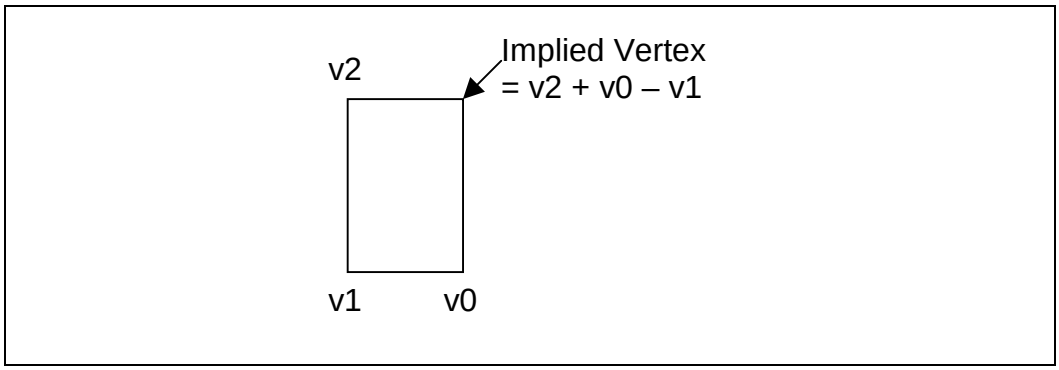


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

7.3.7 Rectangle Completion

This stage of the pipeline applies only to 3DOBJ_RECTANGLE objects. Here the X,Y coordinates of the 4th (upper right) vertex of the rectangle object is computed from the first 3 vertices as shown in the following diagram. The other vertex attributes assigned to the implied vertex (v[3]) are UNDEFINED as they are not used. The Object Setup subfunction will use the values at only the first 3 vertices to compute attribute interpolants used across the entire rectangle.

Figure 7-14. Rectangle Completion



7.3.8 Vertex X,Y Clamping and Quantization

At this stage of the pipeline, vertex X and Y positions are in continuous screen (pixel) coordinates. These positions are quantized to subpixel precision by rounding the incoming values to the nearest subpixel (using round-to-nearest-or-even rules). The device supports rasterization with either 4 or 8 fractional (subpixel) position bits, as specified by the **Vertex SubPixel Precision Select** bit of SF_STATE.

The vertex X and Y screenspace coordinates are also **clamped** to the range [-8K,8K). Note that this clamping occurs after the Drawing Rectangle Origin has been applied and objects have been expanded (i.e., points have been expanded to squares, etc.). In almost all circumstances, if an object's vertices are actually modified by this clamping (i.e., had X or Y coordinates outside of [-8K,8K)) the rendered object will not match the intended result. Therefore software should take steps to ensure that this does not happen – e.g., by clipping objects such that they do not exceed these limits after the Drawing Rectangle is applie.

In addition, in order to be correctly rendered, objects must have a screenspace bounding box not exceeding 8K in the X or Y direction. This additional restriction must also be comprehended by software, i.e., enforced by use of clipping.

7.3.9 Degenerate Object Culling

At this stage of the pipeline, “degenerate” objects are discarded. This operation is automatic and cannot be disabled. (The object rasterization rules would by definition cause these objects to be “invisible” – this culling operation is mentioned here to reinforce that the device implementation optimizes these degeneracies as early as possible).

Degenerate objects are defined in Table 7-6.

Table 7-6. Degenerate Objects

objType	Degenerate Object Definition
3DOBJ_POINT	Two or more corner vertices are coincident (i.e., the radius quantized to zero)
3DOBJ_LINE	The endpoints are coincident
3DOBJ_TRIANGLE	All three vertices are collinear or any two vertices are coincident
3DOBJ_RECTANGLE	Two or more corner vertices are coincident

7.3.10 Degenerate Triangle Culling

The DEBUG ONLY state variable **Zero Pixel Triangle Filter Disable** in SF_STATE controls the removal of triangles that cannot generate any pixels due to the fact that their vertices fall between pixel sample points. Disabling this culling function will not impact the rendered image, though there may be some negative impact on performance.

The DEBUG ONLY state variable **2x2 Pixel Triangle Filter Disable** in SF_STATE controls a more aggressive removal of triangles that cannot generate any pixels. The vertices of very small triangles are examined to see if they will generate any pixels. If not, they are discarded. Disabling this culling function will not impact the rendered image, though there may be some negative impact on performance.

7.3.11 Triangle Orientation (Face) Culling

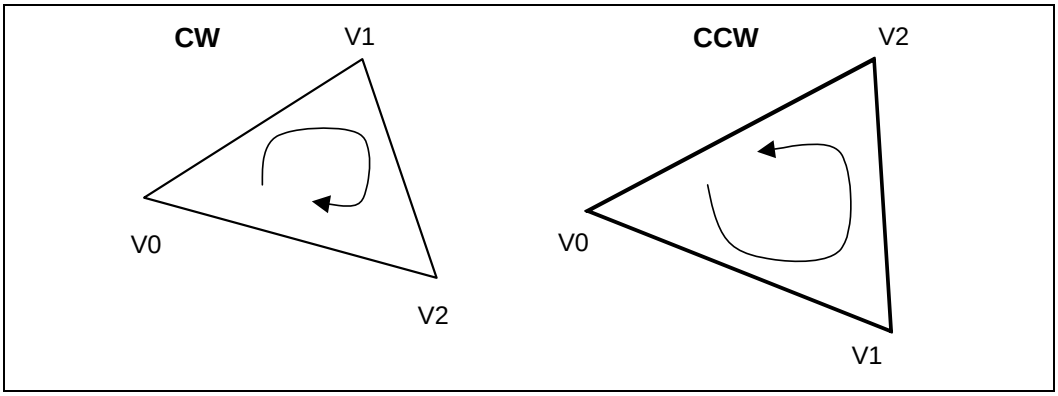
At this stage of the pipeline, 3DOBJ_TRIANGLE objects can be optionally discarded based on the “face orientation” of the object. This culling operation does not apply to the other object types.

This operation is typically called “back face culling”, though front facing objects (or all 3DOBJ_TRIANGLE objects) can be selected to be discarded as well. Face culling is typically used to eliminate triangles facing away from the viewer, thus reducing rendering time.

The “winding order” of a triangle is defined by the the triangle vertex’s 2D (X,Y) screen space position when traversed from v0 to v1 to v2. That traversal will

proceed in either a clockwise (CW) or counter-clockwise (CCW) direction, as shown in Figure 7-15. (A degenerate triangle is considered to have a CW winding order).

Figure 7-15. Triangle Winding Order



The **Front Winding** state variable in SF_STATE controls whether CW or CCW triangles are considered as having a "front-facing" orientation (at which point non-front-facing triangles are considered "back-facing"). The internal variable *invertOrientation* associated with the triangle object is then used to determine whether the orientation of a triangle should be inverted. Recall that this variable is set in the Primitive Decomposition stage to account for the alternating orientations of triangles in strip primitives resulting form the ordering of the vertices used to process them.

The **Cull Mode** state variable in SF_STATE specifies how triangles are to be discarded according to their resultant orientation, as defined in Table 7-6.

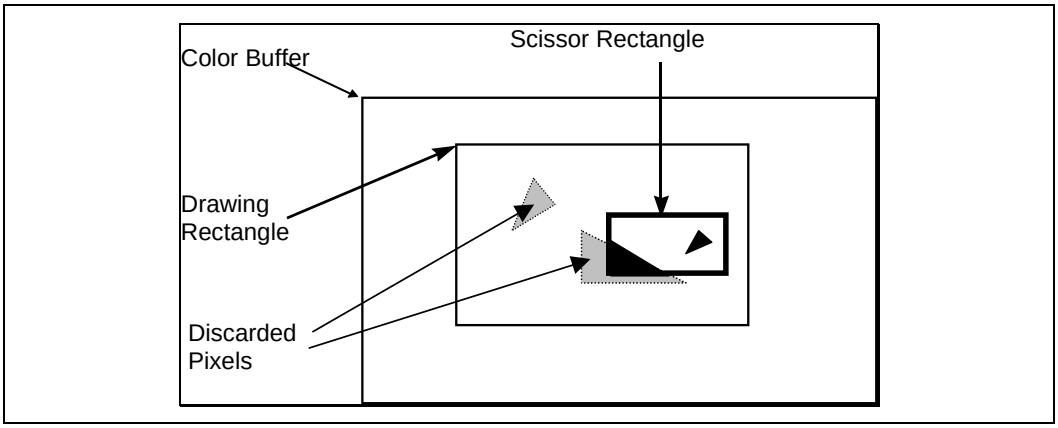
Table 7-7. Cull Mode

CullMode	Definition
CULLMODE_NONE	The face culling operation is disabled
CULLMODE_FRONT	Triangles with "front facing" orientation are discarded
CULLMODE_BACK	Triangles with "back facing" orientation are discarded
CULLMODE_BOTH	All triangles are discarded

7.3.12 Scissor Rectangle Clipping

A *scissor* operation can be used to restrict the extent of rendered pixels to a screen-space aligned rectangle. If the scissor operation is enabled, portions of objects falling outside of the intersection of the scissor rectangle and the clipped draw rectangle are clipped (pixels discarded).

The scissor operation is enabled by the **Scissor Rectangle Enable** state variable in SF_STATE. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding SF_VIEWPORT structure. Up to 16 structures are supported. The **Scissor Rectangle X,Y Min,Max** fields of the SF_VIEWPORT structure defines a scissor rectangle as a rectangle in integer pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. The scissor rectangle is defined



relative to the Drawing Rectangle to better support the OpenGL API. (OpenGL specifies the "Scissor Box" in window-relative coordinates). This allows instruction buffers with embedded Scissor Rectangle definitions to remain valid even after the destination window (drawing rectangle) moves.

The (DEBUG ONLY) **Fast Scissor Clip Disable** state variable in SF_STATE controls how scissor clipping is implemented (though this does not affect the rendered image). If ENABLED (i.e., fast clipping is enabled), only those pixels within the scissor rectangle are rasterized. If DISABLED, the entire object will be rasterized, with object pixels falling outside the scissor rectangle being discarded.

Specifying either scissor rectangle xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null scissor rectangle).

7.3.13 Line Rasterization

The device supports three styles of line rendering: *zero-width* (*cosmetic*) lines, *non-antialiased* lines, and *antialiased* lines. Zero-Width lines are rendered according to the "diamond line rules" Non-antialiased lines are rendered as a polygon having a specified width as measured parallel to the major axis of the line. Antialiased lines are rendered as a rectangle having a specified width measured perpendicular to the line connecting the vertices.

The functions required to render lines is split between the SF and WM units. The SF unit is responsible for computing the overall geometry of the object to be rendered, including the pixel-exact bounding box, edge equations, etc., and therefore is provided with the screen-geometry-related state variables. The WM unit performs the actual scan conversion, determining the exact pixel included/excluded and coverage value for anti-aliased lines.

7.3.13.1 Zero-Width (Cosmetic) Line Rasterization

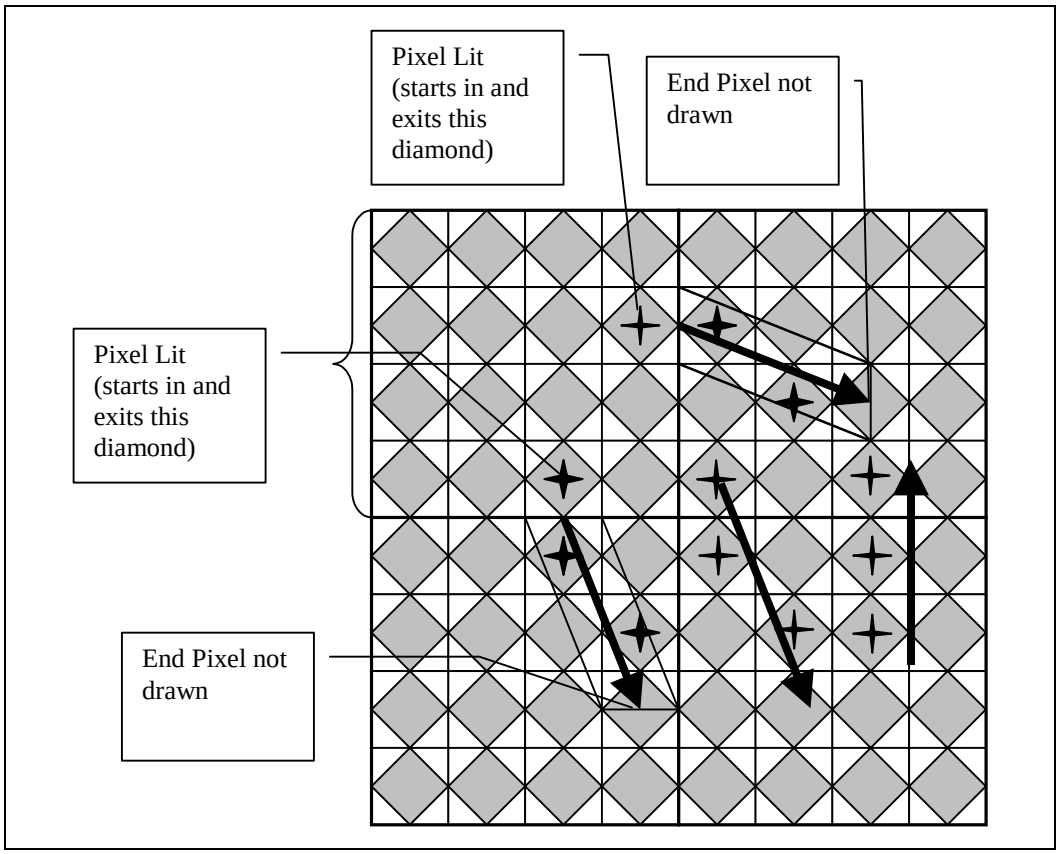
(The specification of zero-width line rasterization would be more correctly included in the WM Unit chapter, though is being included here to keep it with the rasterization details of the other line types).

When the **Line Width** is set to zero, the device will use special rules to rasterize zero-width ("cosmetic") lines. The **Anti-Aliasing Enable** state variable is ignored when **Line Width** is zero.

When the *LineWidth* is set to zero, the device will use special rules to rasterize "cosmetic" lines. The diamond exit rasterization rules comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

The diamond exit rules basically intersect the directed, ideal line connecting two endpoints with an array of diamond-shaped areas surrounding pixel sample points. Wherever the line exits a diamond (including passing through a diamond), the corresponding pixel is lit. Special rules are used to define the subpixel locations which are considered interior to the diamonds, as a function of the slope of the line. When a line ends in a diamond (and therefore does not exit that diamond), the corresponding pixel is not drawn. When a line starts in a diamond and exits that diamond, the corresponding pixel is drawn.

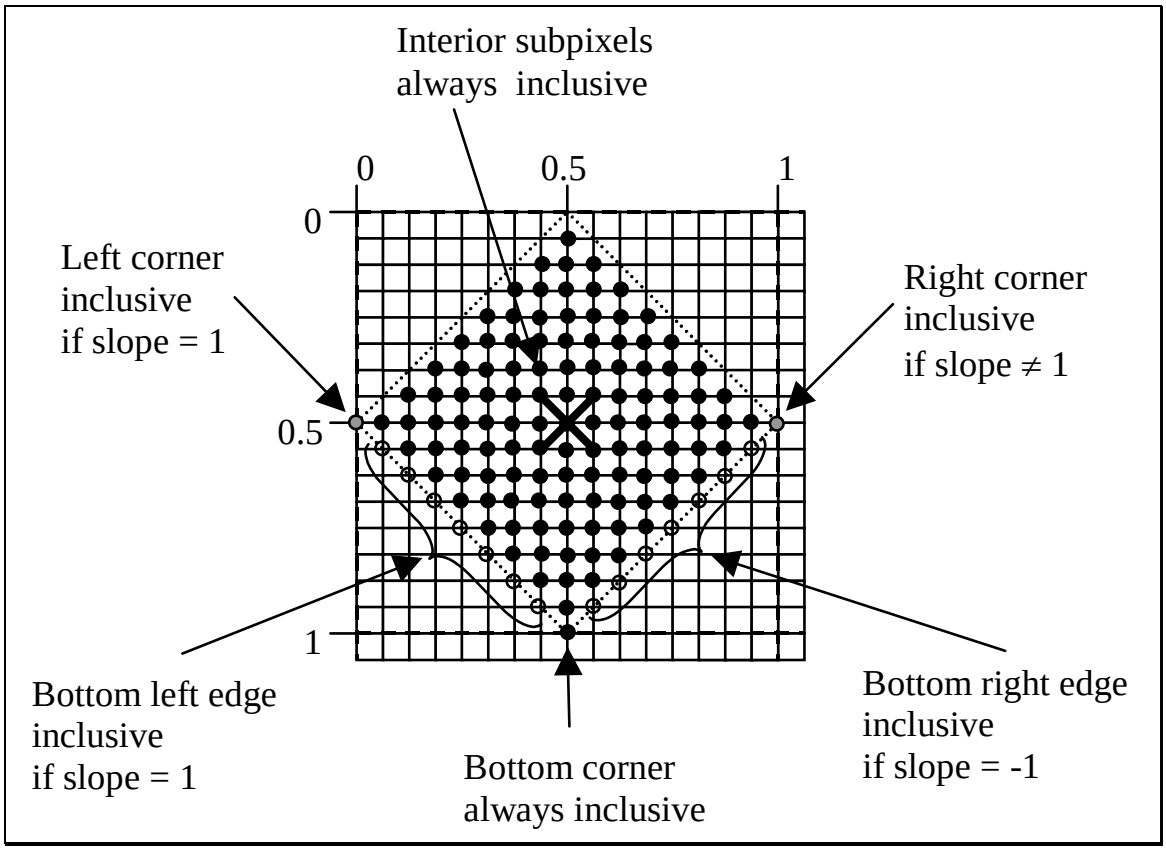
The following diagram shows some examples of diamond exit rendered lines.



The following subsections describe the diamond exit rules in more detail.

7.3.13.2 Diamond Exit Sampling Rules – Legacy Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is ENABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel



of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.

The solid-colored subpixels are considered "interior" to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.

2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range (1, +infinity] or [-infinity, -1) touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) – where if the right corner

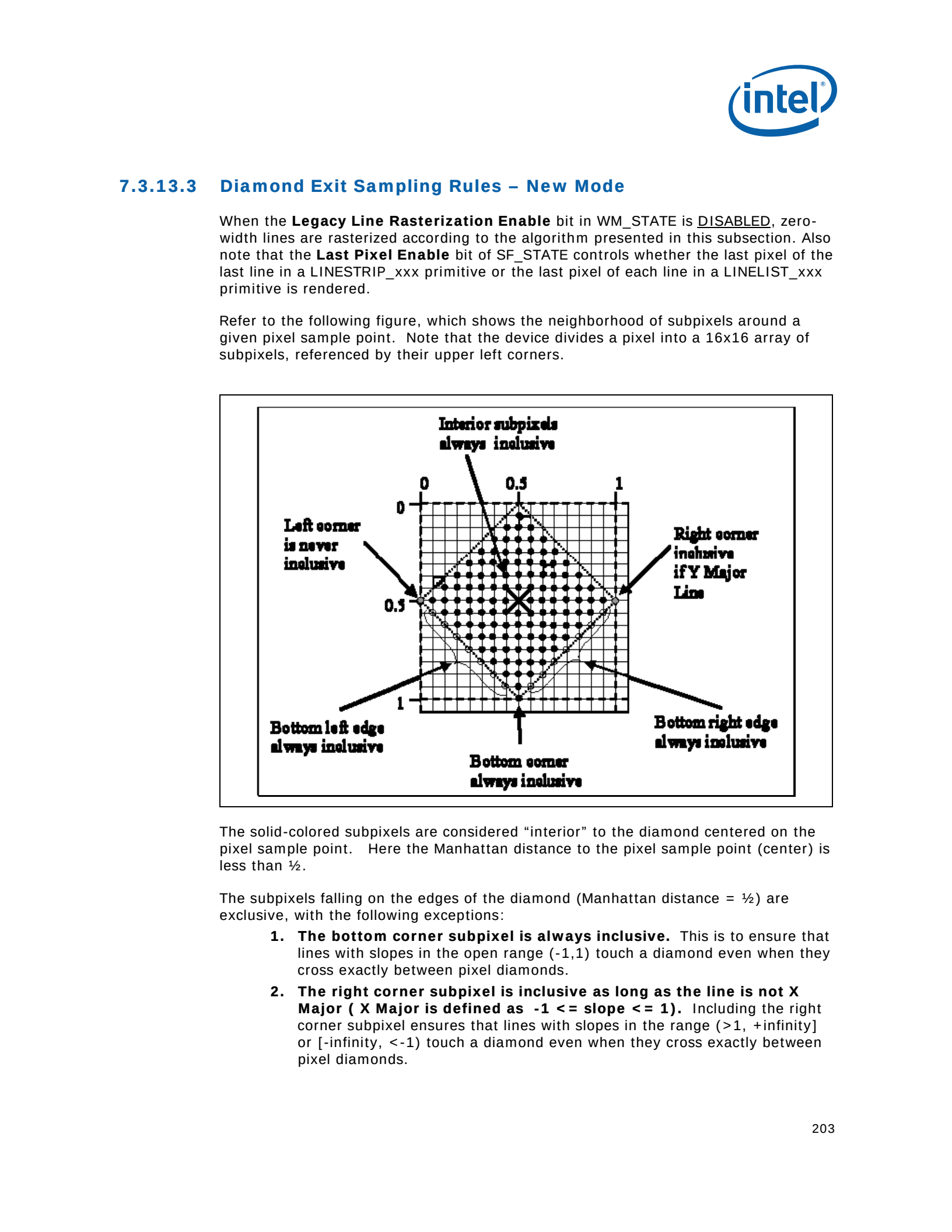
was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).

3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside" for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```
delta_x             = point.x – sample.x
delta_y             = point.y – sample.y
distance            = abs(delta_x) + abs(delta_y)
interior            = (distance < 0.5)
bottom_corner           = (delta_x == 0.0)   && (delta_y == 0.5)
left_corner             = (delta_x == –0.5) && (delta_y == 0.0)
right_corner            = (delta_x == 0.5)   && (delta_y == 0.0)
bottom_left_edge    = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge   = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)
inside = interior ||
                bottom_corner ||
                (slopePosOne ? left_corner : right_corner) ||
                (slopePosOne && left_edge) ||
                (slopeNegOne && right_edge)
```



7.3.13.3 Diamond Exit Sampling Rules – New Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is DISABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.

The solid-colored subpixels are considered “interior” to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as -1 <= slope <= 1).** Including the right corner subpixel ensures that lines with slopes in the range (>1, +infinity] or [-infinity, <-1) touch a diamond even when they cross exactly between pixel diamonds.

3. **The left corner subpixel is never inclusive.** For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
4. **The subpixels along the bottom left edge are always inclusive.** This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside", we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.
5. **The subpixels along the bottom right edge are always inclusive.** Same as case as (4), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (XMajor).

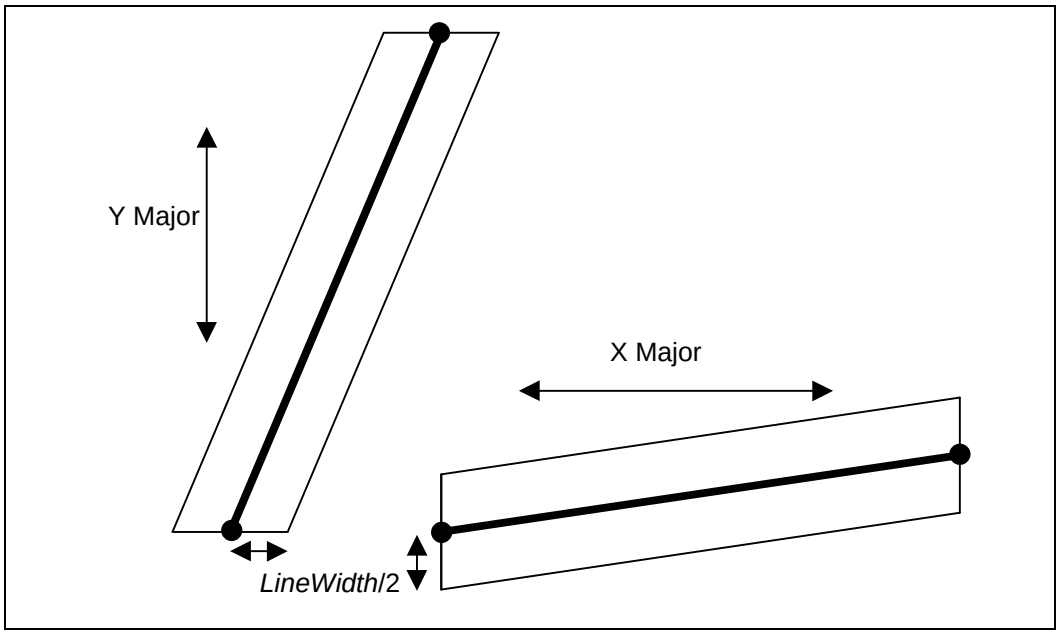
```
delta_x          = point.x – sample.x
delta_y          = point.y – sample.y
distance         = abs(delta_x) + abs(delta_y)
interior         = (distance < 0.5)
bottom_corner        = (delta_x == 0.0)   && (delta_y == 0.5)
left_corner          = (delta_x == –0.5) && (delta_y == 0.0)
right_corner         = (delta_x == 0.5)   && (delta_y == 0.0)
bottom_left_edge     = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge    = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)
inside = interior ||
                bottom_corner ||
                (!XMajor && right_corner) ||
                ( bottom_left_edge) ||
                ( bottom_right_edge)
```

7.3.13.4 Non-Antialiased Wide Line Rasterization

Non-anti-aliased, non-zero-width lines are rendered as parallelograms that are centered on, and aligned to, the line joining the endpoint vertices. Pixels sampled interior to the parallelogram are rendered; pixels sampled exactly on the parallelogram edges are rendered according to the polygon "top left" rules.

The parallelogram is formed by first determining the major axis of the line (diagonal lines are considered x-major). The corners of the parallelogram are computed by translating the line endpoints by +/-(**Line Width** / 2) in the direction of the minor axis, as shown in the following diagram.

Figure 7-16. Non-Antialiased Line Rasterization



7.3.13.5 Anti-aliased Line Rasterization

Anti-aliased lines are rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices. For each pixel in the rectangle, a fractional coverage value (referred to as Antialias Alpha) is computed – this coverage value will normally be used to attenuate the pixel's alpha in the pixel shader thread. The resultant alpha value is therefore available for use in those downstream pixel pipeline stages in order to generate the desired effect (e.g., use the attenuated alpha value to modulate the pixel's color, and add the result to the destination color, etc.). Note that software is required to explicitly program the pixel shader and pixel pipeline to obtain the desired anti-aliasing effect – the device will simply make the coverage-attenuated pixel alpha values available for use in the pixel shader.

The dimensions of the rendered rectangle, and the parameters controlling the coverage value computation, are programmed via the **Line Width**, **Line AA Region**, and **Line Cap AA Region** state variables, as shown below. The edges parallel to the line are located at the distance (*LineWidth*/2) from the line (measured in screen pixel units perpendicular to the line). The end-cap edges are perpendicular to the line and located at the distance (*LineCapAARegion*) from the endpoints.

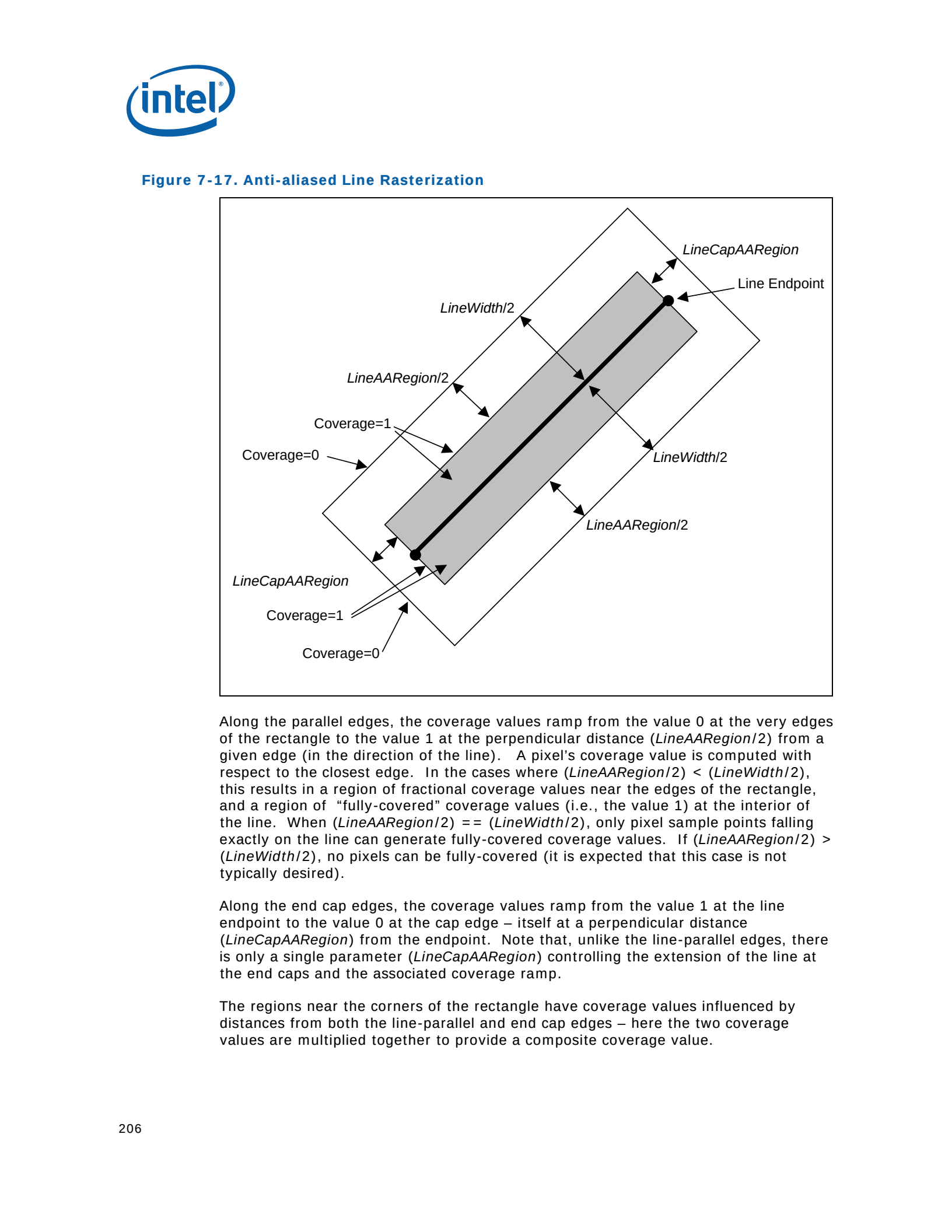


Figure 7-17. Anti-aliased Line Rasterization

Along the parallel edges, the coverage values ramp from the value 0 at the very edges of the rectangle to the value 1 at the perpendicular distance (*LineAARegion*/2) from a given edge (in the direction of the line). A pixel's coverage value is computed with respect to the closest edge. In the cases where (*LineAARegion*/2) < (*LineWidth*/2), this results in a region of fractional coverage values near the edges of the rectangle, and a region of "fully-covered" coverage values (i.e., the value 1) at the interior of the line. When (*LineAARegion*/2) == (*LineWidth*/2), only pixel sample points falling exactly on the line can generate fully-covered coverage values. If (*LineAARegion*/2) > (*LineWidth*/2), no pixels can be fully-covered (it is expected that this case is not typically desired).

Along the end cap edges, the coverage values ramp from the value 1 at the line endpoint to the value 0 at the cap edge – itself at a perpendicular distance (*LineCapAARegion*) from the endpoint. Note that, unlike the line-parallel edges, there is only a single parameter (*LineCapAARegion*) controlling the extension of the line at the end caps and the associated coverage ramp.

The regions near the corners of the rectangle have coverage values influenced by distances from both the line-parallel and end cap edges – here the two coverage values are multiplied together to provide a composite coverage value.

The computed coverage value for each pixel is passed through the Windower Thread Dispatch payload. The Pixel Shader kernel should be passed (unmodified) by the shader to the Render Cache as part of it's output message.

7.3.13.5.1 Anti-aliased Line Distance Mode

The distance from a pixel to the line is approximated by the "Manhattan Distance":

(abs(delta_x)+abs)delta_y

7.4 SF Pipeline State Summary

7.4.1 SF_STATE

The SF_STATE structure defines the layout and function of the data referenced by the **Pointer to SF State** field of the PIPELINE_STATE_POINTERS command.

Note: The majority of the fields in DWords 0-4 have a common definition among the various 3D pipeline FF units. Refer to *3D Pipeline* for a general description of these fields.

DWord	Bit	Description
0	31:6	**Kernel Start Pointer:** This field specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address**. See *3D Pipeline* for more information. **[DevBW-A] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:6]
	5:4	Reserved : MBZ
	3:1	**GRF Register Count:** Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. See *3D Pipeline* for more information. Format = U3 register block count - 1 Range = [0,7] = [16,128] GRF registers
	0	Reserved : MBZ

DWord	Bit	Description
1	31	**Single Program Flow (SPF):** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). The setting of this field must agree with how the kernel program was generated (i.e., single-vs-multiple program flow must be comprehended in the kernel programming, and cannot be accomplished simply through control of this bit.. See CR0 description in *ISA Execution Environment* for more information. 0 = Multiple Program Flows 1 = Single Program Flow
	30:26	Reserved : MBZ
	25:18	**Binding Table Entry Count:** Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** For kernels using a large number of binding table entries, it may be advantageous to set this field to zero to avoid prefetching too many entries and thrashing the state cache. See *3D Pipeline* for more information. Format = U8 Range = [0,255]
	17	**Thread Priority:** Specifies the priority of the thread for dispatch. 0 = Normal Priority 1 = High Priority **Programming Notes:** • this field must be zero.
	16	**Floating Point Mode:** Specifies the floating point mode used by the dispatched thread. 0 = Use IEEE-754 Rules 1 = Use alternate rules
	15:14	Reserved: MBZ
	13	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	12	Reserved: MBZ
	11	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. Format: Enable
	10:8	Reserved: MBZ
	7	**Software Exception Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	6:0	Reserved: MBZ

DWord	Bit	Description
2	31:10	**Scratch Space Base Pointer:** Specifies the 1K-byte aligned offset of the scratch space area allocated to this FF unit. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space**. It is specified as a 1K-byte-granular offset from the **General State Pointer**. Format = GeneralStateOffset[31:10]
	9:4	Reserved : MBZ
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** each get **Per Thread Scratch Space** size without exceeding the driver-allocated scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two **Programming Notes:** • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it.
3	31	Reserved : MBZ
	30:25	**Constant URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. Format = U6 Range = [0,63]
	24	Reserved : MBZ
	23:18	**Constant URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload. Format = U6 Range = [0,63]
	17	Reserved : MBZ
	16:11	**Vertex URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. Programming Notes: It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread. Format = U6 Range = [1,63]
	10	Reserved : MBZ

DWord	Bit	Description
	9:4	**Vertex URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread. Format = U6 Range = [0,63]
	3:0	**Dispatch GRF Start Register for URB Data:** Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload. Format = U4 Range = [0,15]
4	31	Reserved : MBZ
	30:25	**Maximum Number of Threads:** Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. Format = U6 representing thread count - 1 Range = [0,23] indicating thread count of [1,24]
	24	Reserved : MBZ
	23:19	**URB Entry Allocation Size:** Specifies the length of each URB entry used by the unit, in 512-bit register increments - 1. **Programming Note:** Any change to this value requires a subsequent URB_FENCE command (see *Graphics Processing Engine*). Format = U5 Range = [0,31] indicating [1,32] 512-bit register increments
	18:11	**Number of URB Entries:** Specifies the number of URB entries that are used by the unit. **Programming Note:** Any change to this value requires a subsequent URB_FENCE command (see *Graphics Processing Engine*). Format = U8 Range = [1,64]
	10:0	Reserved : MBZ
5	31:5	**Setup Viewport State Offset:** Specifies the 32-byte aligned offset of SF_VIEWPORT. This offset is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** SF_VIEWPORT data pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5]
	4:2	Reserved : MBZ
	1	**Viewport Transform Enable:** This bit controls the Viewport Transform function. Format = Enable

DWord	Bit	Description
	0	**Front Winding:** Determines whether a triangle object is considered "front facing" if the screen space vertex positions, when traversed in the order, result in a clockwise (CW) or counter-clockwise (CCW) winding order. Does not apply to points or lines. 0 = FRONTWINDING_CW 1 = FRONTWINDING_CCW
6	31	**Anti-aliasing Enable:** This field enables "alpha-based" line antialiasing. Format = Enable **Programming Notes:** This field must be disabled if any of the render targets have integer (UINT or SINT) surface format.
	30:29	**Cull Mode:** Controls removal (culling) of triangle objects based on orientation. The cull mode only applies to triangle objects and does not apply to lines, points or rectangles. **Programming Notes:** Orientation determination is based on the setting of the **Front Winding** state. Format = 3D_CullMode: 0 = CULLMODE_BOTH All triangles are discarded (i.e., no triangle objects are drawn) 1 = CULLMODE_NONE No triangles are discarded due to orientation 2: CULLMODE_FRONT Triangles with a front-facing orientation are discarded 3: CULLMODE_BACK Triangles with a back-facing orientation are discarded
	28	**Fast Scissor Clip Disable (DEBUG ONLY) :** This DEBUG ONLY bit can be used to disable the "Fast Scissor Clip" function. When disabled, scissor operations are performed, albeit at lower performance. Format: Disable
	27:24	**Line Width:** Controls width of line primitives. Setting a Line Width of 0.0 specifies the rasterization of the "thinnest" (one-pixel-wide), non-antialiased lines. Note that this effectively overrides the effect of *AAEnable* (though the AAEnable state variable is not modified). Lines rendered with zero Line Width are rasterized using Diamond Exit rules. Format = U3.1 (Units: pixels) Range = [0.0, 7.5]
	23:22	Line End Cap Antialiasing Region Width: **This field specifies the distances over which the coverage of anti-aliased line end caps are computed.** Format = 0 = 0.5 pixels 1= 1.0 pixels 2 = 2.0 pixels 3 = 4.0 pixels Note: this state is duplicated in the windower state descriptor

DWord	Bit	Description
	21:20	**Point Rasterization Rule:** This field specifies the rasterization rules to be applied whenever the edges of a point primitive fall exactly on a pixel sampling point. Format = 3D_RasterizationRule 0 = RASTRULE_UPPER_LEFT – To match "normal" upper left rules for surface primitives 1 = RASTRULE_UPPER_RIGHT – To match OpenGL point rasterization rules (round to + infinity, where this is the upper right direction wrt OpenGL screen origin of lower left). 2 = Reserved (RASTRULE_LOWER_LEFT not seen as useful) 3 = Reserved (RASTRULE_LOWER_RIGHT not seen as useful)
	19	**Zero Pixel Triangle Filter Disable (DEBUG ONLY) :** Disables the culling of some primitives that cannot generate any pixels (otherwise the culling of these primitives is performed when possible). **Programming Notes:** Disabling this filter should not affect the image rendered – only the performance of the device should be impacted. Format = Disable
	18	**2x2 Pixel Triangle Filter Disable (DEBUG ONLY) :** Disables the culling of some zero-pixel triangles that otherwise might be discarded by the "Small Triangle Filter". The 2x2 Triangle Filter attempts to remove triangles with bounding boxes of 2x2 pixels or less and that do not light any pixels. **Programming Notes:** Disabling this filter should not affect the image rendered – only the performance of the device should be impacted. Format = Disable
	17	**Scissor Rectangle Enable:** Enables operation of Scissor Rectangle. Format = Enable
	16:13	**Destination Origin (Pixel Sample Point) Horizontal Bias:** This value is used to specify the horizontal subpixel position of the pixel sampling points (grid) used during rasterization. It is used in conjunction with the vertical bias (below) to position the pixel-sampling grid to provide the rasterization required by the API or the operation at hand. E.g., when rendering triangles, pixels will only be lit when their corresponding sample points fall within the triangle or exactly along certain edges of the triangle – and repositioning the sampling grid will yield somewhat different results. The unbiased sampling points (i.e., when this bias is (0.0,0.0)) are located at the upper-left corner of each screen space pixel. This places the sampling points at the intersections of the integer screen space coordinate grid. Biasing by (0.5.,0.5) positions the pixel sampling points at the center of each screen space pixel (halfway between the integer coordinate grid) – which is typically required to meet OpenGL rasterization rules. Format = U0.4 Range = 0.0 (ox0) or 0.5 (0x8)

DWord	Bit	Description
	12:9	**Destination Origin (Pixel Sample Point) Vertical Bias:** This value is used to specify the vertical position of the pixel sampling points (grid) used during rasterization. (See above description of the horizontal bias). Format = U0.4 Range = 0.0 (ox0) or 0.5 (0x8)
	8:0	Reserved : MBZ
7	31	**Last Pixel Enable:** If ENABLED, the last pixel of a diamond line will be lit. This state will only affect the rasterization of Diamond lines (will not affect wide lines or anti-aliased lines). **Programming Notes:** Last pixel is applied to all lines of a LINELIST, and only the last line of a LINESTRIP. Format = Enable
	30:29	**Triangle Strip/List Provoking Vertex Select:** Selects which vertex of a triangle (in a triangle strip or list primitive) is considered the “provoking vertex”. Used for flat shading of primitives. Format = 0-based vertex index 0h = Vertex 0 1h = Vertex 1 2h = Vertex 2 3h = Reserved
	28:27	**Line Strip/List Provoking Vertex Select:** Selects which vertex of a line (in a line strip or list primitive) is considered the “provoking vertex”. Format = 0-based vertex index 0h – Vertex 0 1h – Vertex 1 2h – Reserved 3h – Reserved
	26:25	**Triangle Fan Provoking Vertex Select:** Selects which vertex of a triangle (in a triangle fan primitive) is considered the “provoking vertex”. Format = 0-based vertex index 0h = Vertex 0 1h = Vertex 1 2h = Vertex 2 3h = Reserved
	24:15	Reserved : MBZ
	14	Reserved: MBZ

DWord	Bit	Description
	13	**Sprite Point Enable:** This bit is passed into the Setup thread payload for use by the Setup kernel as a hint to the setup kernel to overload texture coordinate setup to map some/all texture coordinates to full range (though there is no hardware requirement to do so). Software is free to use this bit for other purposes – it is simply inserted into SF thread payloads. Format: Enable
	12	**Vertex Sub Pixel Precision Select:** Selects the number of fractional bits maintained in the vertex data Format: 0 = 8 sub pixel precision bits maintained 1 = 4 sub pixel precision bits maintained
	11	**Use Point Width State:** Controls whether the point width passed on the vertex or from state is used for rendering point primitives. 0 = Use Point Width on Vertex 1 = Use Pointwidth from State
	10:0	**Point Width:** This field specifies the size (width) of point primitives in pixels. This field is overridden (though not overwritten) whenever point width information is passed in the FVF. Format = U8.3 Range = [0.125, 255.875] pixels

7.4.2 SF_VIEWPORT

The viewport-specific state used by the SF unit (SF_VIEWPORT) is stored as an array of up to 16 elements, each of which contains the DWords described below. The start of each element is spaced 8 DWords apart. The location of first element of the array, as specified by **Setup Viewport State Offset**, is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31:0	**Viewport Matrix Element m00** Format = IEEE_Float
1	31:0	**Viewport Matrix Element m11** Format = IEEE_Float
2	31:0	**Viewport Matrix Element m22** Format = IEEE_Float
3	31:0	**Viewport Matrix Element m30** Format = IEEE_Float
4	31:0	**Viewport Matrix Element m31** Format = IEEE_Float
5	31:0	**Viewport Matrix Element m32** Format = IEEE_Float
6	31:16	**Scissor Rectangle Y Min:** Specifies Y Min coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *less than* **Y Min** will be clipped out if Scissor Rectangle is enabled. NOTE: If Y Min is set to a value greater than Y Max, all primitives will be discarded for this viewport. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]
	15:0	**Scissor Rectangle X Min:** Specifies X Min coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) X coordinates *less than* **X Min** will be clipped out if Scissor Rectangle is enabled. NOTE: If X Min is set to a value greater than X Max, all primitives will be discarded for this viewport. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]
7	31:16	**Scissor Rectangle Y Max:** Specifies Y Max coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *greater than* **Y Max** will be clipped out if Scissor Rectangle is enabled. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]
	15:0	**Scissor Rectangle X Max:** Specifies X Max coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *greater than* **X Max** will be clipped out if Scissor Rectangle is enabled. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]

7.5 The SF Thread -- Interpolation Coefficient Calculation

The final step in object setup is to calculate the interpolation coefficients. This must be done separately (though hopefully in parallel) for each vertex attribute, and is performed by a thread running on an execution unit.

7.5.1 SF Setup Parameters Passed to SF Thread

This section describes some of the parameters computed by the SF unit and passed to SF threads.

7.5.1.1 TRIANGLE Parameters

The SF unit reorders triangle vertices prior to setup computation. The “start vertex” (V0) is defined as being the top-most (least positive Y position) vertex. If more than one vertex shares this Y position, the left-most (least positive Z position) vertex is selected. Once the start vertex is determined, V1 is the next vertex in the clockwise direction, and V2 is the remaining vertex. (Note that degenerate triangles will have been removed by this point, therefore there is no ambiguity in vertex reordering.)

Once the vertices are reordered into V0,V1,V2, the SF unit computes the **Y2-Y0**, **Y1-Y0**, **X2-X0**, **X1-X0**, and **Determinant** values (described in the thread payload below).

The SF unit will use the V0,V1,V2 ordering for the VUE data that follows the thread payload (and possibly the CURBE portion of the payload).

7.5.1.2 RECTANGLE Parameters

With regard to SF thread payload, RECTANGLE objects are handled just like TRIANGLE objects. The 3 vertices supplied for the object are subject to reordering and used in SF unit setup computations. The same parameters are passed in the thread payload as for TRIANGLE objects, and the 3 (possibly reordered) VUEs are included in the payload.

7.5.1.3 POINT Parameters

Point width is applied to POINT objects, expanding them to screen-aligned squares. The SF unit selects the following vertices for the normal setup computations: Upper-left = V0, Lower-right = V1, Lower-left = V2. In this respect they appear as RECTANGLES in the SF thread payload. However, only the single original object vertex (the center) is passed as VUE data.

The **Sprite Point Enable** bit from SF_STATE is passed in the SF thread header to assist in the support of API “sprite points,” where some/all texture coordinates are set to full-range over the point square vs. all corners being assigned the constant value provided by the object (center) vertex.

7.5.1.4 LINE Parameters

The SF unit reorders line vertices prior to setup computation. The "start vertex" (V0) is defined as being the top-most (least positive Y position) vertex. If the other vertex shares this Y position, the left-most (least positive Z position) vertex is selected. Once the start vertex is determined, V1 is the remaining vertex. (Note that degenerate lines will have been removed by this point, therefore there is no ambiguity in vertex reordering.)

Once the vertices are reordered into V0,V1, the SF unit computes the **Y1-Y0**, **X1-X0**, and **Determinant** values (described in the thread payload below).

The SF unit will use the V0,V1 ordering for the VUE data that follows the thread payload (and possibly the CURBE portion of the payload).

7.5.2 SF (Setup) Thread Payload

DWord	Bit	Description
R0.7	31	**Snapshot Flag:** If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID:**This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. (See *Debuging* for further description). Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer:**Specifies the 1K-byte aligned offset (from the **General State Base Address**) to the scratch space allocated to this thread. Format = GeneralStateOffset[31:10]
	9:8	Reserved
	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. Format: U8
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:0	Reserved

DWord	Bit	Description
R0.1	31:0	Reserved
R0.0	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.
	15:0	**URB Return Handle:** This is the URB handle where the thread's results are to be placed (aka the Primitive URB Entry, or PUE).
R1.7	31:0	Reserved
R1.6	31:0	Y2-Y0 (aka dY2) : For TRIANGLE, RECT and POINT objects: This field contains the value (Y2 – Y0) , where the indices are relative to the "start" vertex. This value is also known as "dY2" , where the "2" is the relative order of the delta term around a triangle, not a vertex index. For LINE objects: Reserved Format: FLOAT32
R1.5	31:0	**Y1-Y0** (aka dY0) **:** For all objects: This field contains the value (Y1 – Y0) , where the indices are relative to the "start" vertex. This value is also known as "dY0" , where the "0" is the relative order of the delta term around a triangle, not a vertex index. Format: FLOAT32
R1.4	31:0	X2-X0 (aka dX2) : For TRIANGLE, RECT and POINT objects: This field contains the value (X2 – X0) , where the indices are relative to the "start" vertex. This value is also known as "dX2" , where the "2" is the relative order of the delta term around a triangle, not a vertex index. For LINE objects: Reserved Format: FLOAT32
R1.3	31:0	X1-X0 (aka dX0) : For all objects: This field contains the value (X1 – X0) , where the indices are relative to the "start" vertex. This value is also known as "dX0" , where the "0" is the relative order of the delta term around a triangle, not a vertex index. Format: FLOAT32
R1.2	31:0	Determinant For TRIANGLE, RECT and POINT objects: (X1-X0)(Y2-Y0) – (X2-X0)(Y1-Y0) For LINE objects: (X1-X0)(X1-X0) + (Y1-Y0)(Y1-Y0) Format: FLOAT32
R1.1	31:0	**Provoking Vertex:** This field contains the relative index (0-2) of the reordered vertex considered the "provoking" vertex, given the PrimType and related SF_STATE state variables (**xxx Provoking Vertex Select**). The SF thread can use this value when performing setup computations for "constant-interpolated" vertex attributes. 0 = V0 1 = V1 2 = V2
R1.0	31:18	Reserved

DWord	Bit	Description
	17	**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
	16	**Sprite Point Enable:**This a copy of the **Sprite Point Enable** bit in SF_STATE. It is passed in the payload strictly for use by the SF (Setup) thread – there is no other hardware function involved. For example (and the expected usage model), a setup kernel processing a point object could overload texture coordinate setup to map texture to full range, thus mapping a texture to the sprite point. Format: Enable
	15:0	**Primitive Type:** This is the unmodified PrimType of the primitive topology containing the object, as received from the 3D pipeline. E.g., a point object within a POINTLIST will have POINTLIST passed in this field even though the point is expanded to a square. Format: See 3DPRIMITIVE description in *Vertex Fetch* for encoding
R2.7	31:0	Reserved
R2.6	31:0	Reserved
R2.5	31:0	Inverse W2: For TRIANGLE, RECTANGLE and POINT objects: This is the position 1/W value associated with V2. The SF thread can use this value (passed directly from the SF unit) in order to avoid having to have the Vertex Header portions of the object vertex VUEs from being included in the VUE portion of the SF thread payload. For LINE objects: Reserved Format: FLOAT32
R2.4	31:0	Z2: For TRIANGLE, RECTANGLE and POINT objects: This is the position Z value associated with V2. The SF unit computes this value given the position Z value from the VUE Vertex Header and state information, etc. For LINE objects: Reserved Format: FLOAT32
R2.3	31:0	Inverse W1 : For all objects: This is the position 1/W value associated with V1. See **Inverse W2.** Format: FLOAT32
R2.2	31:0	Z1 For all objects: This is the position Z value associated with V1. See **Z2.** Format: FLOAT32
R2.1	31:0	Inverse W0 For all objects: This is the position 1/W value associated with V0. See **Inverse W2.** Format: FLOAT32

DWord	Bit	Description
R2.0	31:0	Z0 For all objects: This is the position Z value associated with V0. See **Z2.** Format: FLOAT32
[varies]	31:0	Constant Data from CURBE URB Entry (optional)
[varies]	31:0	V0 Vertex Attribute (VUE) Data from URB (for all objects)
[varies]	31:0	V1 Vertex Attribute (VUE) Data from URB (for all objects except POINTs)
[varies]	31:0	V2 Vertex Attribute (VUE) Data from URB (for TRIANGLE and RECTANGLE objects only)

7.5.3 SF Thread Execution

The kernel that performs coefficient interpolation must be supplied by the jitter. As a usage note, it generally needs to loop through the entire set of vertex attributes, calculating a C0, Cx and Cy for each. It must take into account whether or not “wrap shortest” mode is on, if flat (rather than gouraud) shading has been selected, whether (separately for each attribute) interpolation should be done in a perspective correct manner, if point sprites are enabled, and must operate appropriately for the primitive type (triangle, line or point.)

7.5.4 SF Thread Output

The SF thread must send a URB_WRITE to the URB shared function in order to pass results for use in subsequent PS threads spawned in the rasterization of the object. This information will be read from the URB as part of WM thread dispatch and thus included in the WM thread payload.

DWord	Bit	Description
M1.7	31:0	Cx[7] Gradient in X for attribute 7. Format = IEEE_Float
M1.6	31:0	Cx[6]
M1.5	31:0	Cx[5]
M1.4	31:0	Cx[4]
M1.3	31:0	Cx[3]
M1.2	31:0	Cx[2]
M1.1	31:0	Cx[1]
M1.0	31:0	Cx[0]
M2.7	31:0	Cy[7] Gradient in Y for attribute 7. Format = IEEE_Float
M2.6	31:0	Cy[6]
M2.5	31:0	Cy[5]

DWord	Bit	Description
M2.4	31:0	Cy[4]
M2.3	31:0	Cy[3]
M2.2	31:0	Cy[2]
M2.1	31:0	Cy[1]
M2.0	31:0	Cy[0]
M3.7	31:0	Co[7] Value of attribute 7 at the start vertex (V0) Format = IEEE_Float
M3.6	31:0	Co[6]
M3.5	31:0	Co[5]
M3.4	31:0	Co[4]
M3.3	31:0	Co[3]
M3.2	31:0	Co[2]
M3.1	31:0	Co[1]
M3.0	31:0	Co[0]
M4...		Additional attributes Additional attributes beyond the first 8 are sent in subsequent message registers following the same format as the first 8.

The message descriptor of this URB_WRITE message should set **Swizzle Control** to URB_TRANSPOSE in order to re-arrange the interpolation coefficients by attribute instead of by coefficient type (C0, Cx and Cy) as shown above. See *URB* chapter. This functionality is provided as a performance enhancement; the coefficient interpolation code could send the coefficients in the desired format, but having it re-arrange the coefficients is not as efficient as relying on this hardware mechanism.

Assuming the interpolation coefficient generation thread sent the preceding message with *SF to Windower transpose* swizzle, the resulting URB contents would look like this:

Co3	null	Cy3	Cx3	Co2	null	Cy2	Cx2	Co1	null	Cy1	Cx1	Co0	null	Cy0	Cx0
Co7	null	Cy7	Cx7	Co6	null	Cy6	Cx6	Co5	null	Cy5	Cx5	Co4	null	Cy4	Cx4

This is the most efficient arrangement for the windower interpolation code (“jitted” code placed before the pixel shader).

Note: In order for the WM unit to read back Z plane equation coefficients (as it interpolates Z), the Setup thread must have those coefficients stored in the low-order 4 DWs of a URB row (corresponding to an even-numbered attribute in the diagram above).

7.6 Other SF Functions

7.6.1 Statistics Gathering

The SF stage itself does not have any associated pipeline statistics; however, it counts the number of objects being output by the clipper on the clipper's behalf, since it less feasible to have the CLIP unit figure out how many objects have been output by a clip thread. It is easy for the SF unit to count the number of objects it receives from the CLIP stage since it is decomposing the output primitive topologies into objects anyway.

If the **Statistics Enable** bit is set in SF_STATE, then SF will increment the CL_PRIMITIVES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) once for each object in each primitive topology it receives from the CLIP stage. This bit should always be set if clipping is enabled and pipeline statistics are desired.

Software should always clear the **Statistics Enable** bit in SF_STATE if the clipper is disabled since objects SF receives are not considered "primitives output by the clipper" unless the clipper is enabled. Note that the clipper can be disabled either using bypass mode via a PIPELINE_STATE_POINTERS command with **Clip Enable** clear *or* by setting **Clip Mode** in CLIP_STATE to CLIPMODE_ACCEPT_ALL.

§§

8 Windower (WM) Stage

8.1 Overview

As mentioned in the *SF Unit* chapter, the SF stage prepares an object for scan conversion by the Window/Masker (WM) unit. Refer to the *SF Unit* chapter for details on the screen-space geometry of objects to be rendered. The WM unit uses the parameters provided by the SF unit in the object-specific rasterization algorithms.

The WM stage of the GEN4 3D pipeline performs the following operations (at a high level)

- Pre-scan-conversion modification of some primitive attributes, including
  - Application of Depth Offset to the position Z attribute
- Scan-conversion of the various primitive types, including
  - 2D clipping to the scissor/draw rectangle intersection
- Spawning of Pixel Shader (PS) threads to process the pixels resulting from scan-conversion

The spawned Pixel Shader (PS) threads are responsible for the following (high-level) operations

- Interpolating vertex attributes (other than X,Y,Z) to the pixel location
- Performing any “Pixel Shader” operations dictated by the API PS program
  - Using the Sampler shared function to sample data from “texture” surfaces
  - Using the DataPort to perform general memory I/O
- Submitting the shaded pixel results to the DataPort for any subsequent “blending” (aka Output Merger) operation and write to the RenderCache.

The WM unit keeps a scoreboard of pixels being processed in outstanding PS threads in order to guarantee in-order rasterization results. This allows the WM unit to overlap processing of several objects.

8.1.1 Inputs from SF to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects. The types of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- Handle of the Primitive URB Entry (PUE) that was written by the SF (Setup) thread. This handle will be passed to all WM (PS) threads spawned from the WM's rasterization process.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.)
- Edge or line interpolation information (e.g., edge equation coefficients, etc.)
- Information on where the WM is to start rasterization of the object
- Object orientation (front/back-facing)
- Last Pixel indication (for line drawing)

8.2 Windower Pipelined State

8.2.1 WM_STATE

DWord	Bit	Description
0	31:6	**Kernel Start Pointer[0]:** Specifies the 64-byte aligned address offset of the first instruction in the kernel[0]. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:6]
	5:4	Reserved : MBZ
	3:1	**GRF Register Count[0]:** Defines the number of GRF Register Blocks used by the kernel[0]. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. Format = U3 register block count - 1 Range = [0,7] corresponding to [1,8] 16-register blocks
	0	Reserved : MBZ

DWord	Bit	Description
1	31	**Single Program Flow (SPF) :** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*. 0 = Multiple Program Flows 1 = Single Program Flow
	30:26	Reserved : MBZ
	25:18	**Binding Table Entry Count:** Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** for kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache. Format = U8 Range = [0,255]
	17	**Thread Priority:** Specifies the priority of the thread for dispatch 0 = Normal Priority 1 = High Priority **Programming Notes:** • this field must be zero.
	16	**Floating Point Mode:** Specifies the floating point mode used by the dispatched thread. 0 = Use IEEE-754 Rules 1 = Use alternate rules
	15:14	Reserved : MBZ
	13:8	**Depth Coefficient URB Read Offset:** Specifies the offset (in 256-bit units) at which the depth coefficient URB data is to be read from the URB and used by the FF to interpolate depth. The WM unit interprets the low order 128 bits of this URB row as containing the plane coefficients of Z depth. This places a restriction on the Setup thread to write the URB in such a way as to place these coefficients in the low order DWords of a URB row. See *Strip and Fan Unit* and *URB* chapters for details on Setup threads and the TRANSPOSED URB write operation. Format = U6 Range = [0,63]
	7:5	Reserved : MBZ
	4	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	3	Reserved : MBZ
	2	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. Format: Enable

DWord	Bit	Description
	1	**Software Exception Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	0	Reserved : MBZ
2	31:10	**Scratch Space Base Pointer:** Specifies the 1k-byte aligned address offset to scratch space for use by the kernel. This pointer is relative to the **General State Base Address**. **Programming Note:** • **[DevBW-A] A0 Erratum BWT005**: If **Per Thread Scratch Space** is programmed to 256KB, this pointer must be 8M-aligned. Format = GeneralStateOffset[31:10]
	9:4	Reserved : MBZ
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by each thread. The driver must allocate enough contiguous scratch space, pointed to by the Scratch Space Pointer, to ensure that the Maximum Number of Threads each get Per Thread Scratch Space size without exceeding the driver-allocated scratch space. Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two **Programming Note:** • **[DevBW-A] A0 Erratum BWT005**: The range [0,11] for this register indicates [1KB, 12KB] in 1K byte increments. If MMIO register 21D0h bit 3 is set, then value 11 is an exception and indicates a 256KB space instead of 12KB. Note that Scratch Space Base Pointer must be 8MB-aligned in order to set the 256KB scratch space. Format = U4
3	31	Reserved : MBZ
	30:25	**Constant URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. Format = U6 Range = [0,63]
	24	Reserved : MBZ
	23:18	**Constant URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload. Format = U6 Range = [0,63]

DWord	Bit	Description
	17:11	**Setup URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for each Setup URB entry, in 256-bit register increments. **Programming Notes:** • It is UNDEFINED to set this field to 0 indicating no Setup URB data to be read and passed to the PS thread. Format = U7 Range = [1,63]
	10	Reserved : MBZ
	9:4	**Setup URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Setup URB data is to be read from the URB before being included in the thread payload. This offset applies to all Setup URB entries passed to the thread. Format = U6 Range = [0,63]
	3:0	**Dispatch GRF Start Register for URB Data:** Specifies the starting GRF register number for the URB portion (Constant + Setup) of the thread payload. Format = U4 Range = [0,15] **[DevBW,DevCL]**: If 32 pixel dispatch is enabled, the maximum range is [0,7]
4	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5]
	4:2	**Sampler Count:** Specifies how many samplers (in multiples of 4) the vertex shader 0 kernel uses. Used only for prefetching the associated sampler state entries. Format = U3 Range = [0,4] 0 = no samplers used 1 = between 1 and 4 samplers used 2: between 5 and 8 samplers used 3: between 9 and 12 samplers used 4: between 13 and 16 samplers used
	1	Reserved : MBZ

DWord	Bit	Description
	0	**Statistics Enable:** If ENABLED, the Windower will engage in statistics gathering. If DISABLED, statistics information associated with this FF stage will be left unchanged. See *Statistics Gathering*. **Programming Notes:** • If this field is enabled, **Statistics Enable** in CC_STATE should also be set, and when this field is disabled, **Statistics Enable** in CC_STATE should also be clear. Both functions contribute to the PS_DEPTH_COUNT, so having either one set without the other set will result in an UNPREDICTABLE value for PS_DEPTH_COUNT. • **[DevBW-A] A0 Erratum BWT004**: If no pixel shader is desired (a "null" pixel shader), this bit must be *cleared* so that PS_INVOCATIONS will not be incremented for the "dummy" PS dispatches. Format = Enabled
5	31:25	**Maximum Number of Threads:** Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. Format = U7 representing (thread count – 1) Range = [0, n-1] where n = (# EUs) * (# threads/EU). See *Graphics Processing Engine* for listing of #EUs and #threads in each device.
	24	Reserved : MBZ
	23	**Legacy Diamond Line Rasterization:** This bit, if ENABLED, indicates that the Windower will rasterize zero width lines using the legacy rasterization rules. If DISABLED, the Windower will rasterize zero width lines using the new rasterization rules (see *Strips Fans* chapter). Format = Enable
	22	**Pixel Shader Kill Pixel:** This bit, if ENABLED, indicates that the PS kernel has the ability to kill (discard) pixels, e.g., as required by the presence of a "killpix" or "discard" instruction in the API PS program, or JITTER-introduced code to kill pixels due to ClipDistance clipping. If DISABLED, the PS kernel may not, under any circumstances, kill pixels. This bit must also be ENABLED if a sampler has chroma key enabled with kill pixel mode. Format = Enable
	21	**Pixel Shader Computed Depth:** This bit, if ENABLED, indicates that the PS kernel computes a depth value. It is used to disable the depth/stencil test in the Early Depth Test function. Format = Enable **Programming Notes:** • If a NULL Depth Buffer is selected, the **Pixel Shader Computed Depth** field must be set to disabled. • **[DevBW-A] Errata:** If both **Depth Test Enable** and **Depth Write Enable** are disabled, this field must be disabled.
	20	**Pixel Shader Uses Source Depth:** This bit, if ENABLED, indicates that the PS kernel requires the source depth value (vPos.z) to be passed in the payload. Format = Enable

DWord	Bit	Description
	19	**Thread Dispatch Enable:** This bit, if set, indicates that it is possible for a PS thread to modify a render target, i.e., at least one render target is enabled (is not of type SURFTYPE_NULL and has at least one channel enabled for writes) and the PS kernel contains a code path that may issue a write to that/those enabled RTs. **Programming Notes:** • This bit is used for performance optimizations and does not directly control writing to render targets. If this bit is DISABLED, no pixel shader threads will be dispatched.. • For correct behavior, this bit must be set consistently with the behavior of the PS kernel, i.e. if this bit is DISABLED the PS kernel must not write color or depth to any render targets. Format = Enable
	18	**Early Depth Test Enable:** This bit enables the Early Depth Test (aka Intermediate Z, or IZ) function. Note: This bit should always be ENABLED – at least there are no known conditions underwhich disabling the Early Depth Test is required. Format = Enable
	17:16	**Line End Cap Antialiasing Region Width:** This field specifies the distances over which the coverage of anti-aliased line end caps are computed. Format = 0 = 0.5 pixels 1 = 1.0 pixels 2 = 2.0 pixels 3 = 4.0 pixels **Note:** This state is duplicated in the SF_STATE state descriptor
	15:14	**Line Antialiasing Region Width:** This field specifies the distance over which the anti-aliased line coverage is computed. Format = 0 = 0.5 pixels 1 = 1.0 pixels 2 = 2.0 pixels 3 = 4.0 pixels
	13	**Polygon Stipple Enable:** Enables the Polygon Stipple function. Format = Enable
	12	**Global Depth Offset Enable:** Enables computation and application of Global Depth Offset. Format = Enable
	11	**Line Stipple Enable:** Enables the Line Stipple function. Format = Enable

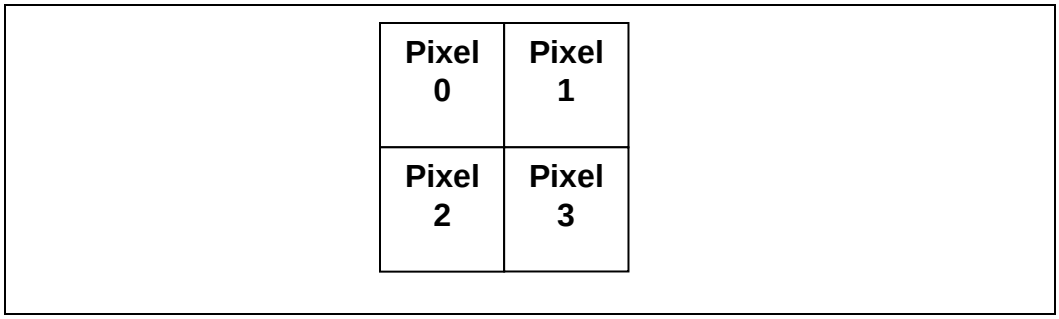
DWord	Bit	Description
	10	**Legacy Global Depth Bias Enable:** Enables the Windower to use the Global Depth Offset Constant state unmodified. If this bit is not set, the Windower will scale the Global Depth Offset Constant as described in section 1.4.2 of this document. Format = Enable
	9	Reserved : MBZ
	8	Reserved : MBZ
	7	Reserved : MBZ
	6:5	Reserved : MBZ
	4	Reserved : MBZ
	3	Reserved : MBZ
	2	**32 Pixel Dispatch Enable:** Enables the Windower to dispatch 8 subspans in one payload 0 = 32 pixel dispatch disabled 1 = 32 pixel dispatch enabled Note: See Table 8-1 for valid pixel dispatch combinations.
	1	**16 Pixel Dispatch Enable:** Enables the Windower to dispatch 4 subspans in one payload (typical operation) 0 = 16 pixel dispatch disabled 1 = 16 pixel dispatch enabled Note: See Table 8-1 for valid pixel dispatch combinations.
	0	**8 Pixel Dispatch Enable:** Enables the Windower to dispatch 2 subspans in one payload 0 = 8 pixel dispatch disabled 1 = 8 pixel dispatch enabled Note: See Table 8-1 for valid pixel dispatch combinations.
6	31:0	**Global Depth Offset Constant:** Specifies the constant term in the GlobalDepthOffset function. Format = IEEE_FP
7	31:0	**Global Depth Offset Scale:** This field specifies the *GlobalDepthOffsetScale* term used in the Global Depth Offset Function Format = IEEE_FP

8.3 Rasterization

The WM unit uses the setup computations performed by the SF unit to rasterize objects into the corresponding set of pixels. Most of the controls regarding the screen-space geometry of rendered objects are programmed via the SF unit.

The rasterization process generates pixels in 2x2 groups of pixels called *subspans* (see Figure 8-1) which, after being subjected to various inclusion/discard tests, are grouped and passed to spawned Pixel Shader (PS) threads for subsequent processing. Once these PS threads are spawned, the WM unit provides only bookkeeping functions on the pixels. Note that the WM unit can proceed on to rasterize subsequent objects while PS threads from previous objects are still executing.

Figure 8-1. Pixels with a SubSpan



8.3.1 Drawing Rectangle Clipping

The Drawing Rectangle defines the maximum extent of pixels which can be rendered. Portions of objects falling outside the Drawing Rectangle will be clipped (pixels discarded). Implementations will typically discard objects falling completely outside of the Drawing Rectangle as early in the pipeline as possible. There is no control to turn off Drawing Rectangle clipping – it is unconditional.

For the purposes of clipping, the Drawing Rectangle must itself be clipped to the destination buffer extents. (The Drawing Rectangle Origin, used to offset relative X,Y coordinates earlier in the pipeline, is permitted to lie offscreen). The **Clipped Drawing Rectangle X,Y Min,Max** state variables (programmed via 3DSTATE_DRAWING_RECTANGLE – See *SF Unit*) defines the intersection of the Drawing Rectangle and the Color Buffer. It is specified with non-negative integer pixel coordinates relative to the Destination Buffer upper-left origin.

Pixels with coordinates outside of the Drawing Rectangle cannot be rendered (i.e., the rectangle is inclusive). For example, to render to a full-screen 1280x1024 buffer, the following values would be required: Xmin=0, Ymin=0, Xmax=1279 and Ymax=1023.

For “full screen” rendering, the Drawing Rectangle coincides with the screen-sized buffer. For “front-buffer windowed” rendering it coincides with the destination “window”.

8.3.2 Line Rasterization

See *SF Unit* chapter for details on the screen-space geometry of the various line types.

8.3.2.1 Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

8.3.2.2 Line Stipple

Line stipple, controlled via the **Line Stipple Enable** state variable in WM_STATE, discards certain pixels that are produced by non-AA line rasterization.

The line stipple rule is specified via the following state variables programmed via 3DSTATE_LINE_STIPPLE: the 16-bit **Line Stipple Pattern** (p), **Line Stipple Repeat Count** I, and **Line Stipple Inverse Repeat Count.** Sofware must compute **Line Stipple Inverse Repeat Count** as 1.0f / **Line Stipple Repeat Count** and then converted from float to the required fixed point encoding (see 3STATE_LINE_STIPPLE).

The WM unit maintains an internal Line Stipple Counter state variable (s). The initial value of s is zero; s is incremented after production of each pixel of a line segment (pixels are produced in order, beginning at the starting point and working towards the ending point). s is reset to 0 whenever a new primitive is processed (unless the primitive type is LINESTRIP_CONT or LINESTRIP_CONT_BF), and before every line segment in a group of independent segments (LINELIST primitive).

During the rasterization of lines, the WM unit computes:

$$b = \lfloor s/r \rfloor \bmod 16,$$

A pixel is rendered if the b^{th} bit of p is 1, otherwise it is discarded. The bits of p are numbered with 0 being the least significant and 15 being the most significant.

8.3.2.3 3DSTATE_LINE_STIPPLE

3DSTATE_LINE_STIPPLE			
Project:	All	**Length Bias:**	2
The 3DSTATE_LINE_STIPPLE command is used to specify state variables used in the Line Stipple function.			

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 08h 3DSTATE_LINE_STIPPLE Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31	**Modify Enable (Current Repeat Counter, Current Stipple Index)** Project: All Format: Enable FormatDesc Modify enable for **Current Repeat Counter** and **Current Stipple Index** fields. **Programming Notes** Software should never set this field to enabled. It is provided only for HW-generated commands as part of context save/restore.
	30	**Reserved** Project: All Format: MBZ
	29:21	**Current Repeat Counter** Project: All Format: U9 FormatDesc This field sets the HW-internal repeat counter state. Note: Software should never attempt to set this value – this state is only provided for HW-generated commands as part of context save/restore.
	20	**Reserved** Project: All Format: MBZ

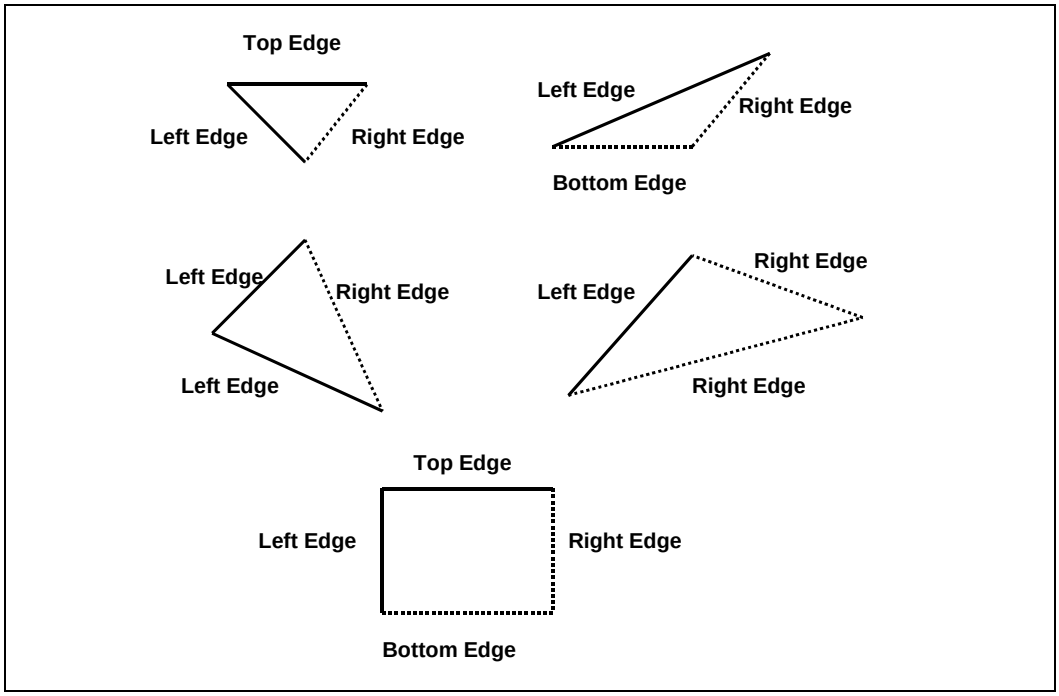
3DSTATE_LINE_STIPPLE		
	19:16	**Current Stipple Index** Project: All Format: U4 FormatDesc This field sets the HW-internal stipple pattern index. Note: Software should never attempt to set this value – this state is only provided for HW-generated commands as part of context save/restore.
	15:0	**Line Stipple Pattern** Project: All Format: 16 bit mask. Bit 15 = most significant bit, Bit 0 = least significant bit FormatDesc Specifies a pattern used to mask out bit specific pixels while rendering lines.
2	31:16	**Line Stipple Inverse Repeat Count** Project: All Format: U1.13 FormatDesc Range [0.00390625, 1.0] Specifies the inverse (truncated) of the repeat count for the line stipple function.
	15:9	**Reserved** Project: All Format: MBZ
	8:0	**Line Stipple Repeat Count** Project: All Format: U9 FormatDesc Range [1, 256] Specifies the repeat count for the line stipple function.

8.3.3 Polygon (Triangle and Rectangle) Rasterization

The rasterization of LINE, TRIANGLE, and RECTANGLE objects into pixels requires a "pixel sampling grid" to be defined. This grid is defined as an axis-aligned array of pixel sample points spaced exactly 1 pixel unit apart. If a sample point falls within one of these objects, the pixel associated with the sample point is considered "inside" the object, and information for that pixel is generated and passed down the pipeline.

For TRIANGLE and RECTANGLE objects, if a sample point intersects an edge of the object, the associated pixel is considered "inside" the object if the intersecting edge is a "left" or "top" edge (or, more exactly, the intersected edge is not a "right" or "bottom" edge). Note that "top" and "bottom" edges are by definition exactly horizontal. The following diagram identifies the edge types for representative TRIANGLE and RECTANGLE objects (solid edges are inclusive, dashed edges are exclusive).

Figure 8-2. TRIANGLE and RECTANGLE Edge Types



8.3.3.1 Polygon Stipple

The *Polygon Stipple* function, controlled via the **Polygon Stipple Enable** state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered. Polygon stipple is applied only to the following primitive types:

3DPRIM_POLYGON
3DPRIM_TRIFAN
3DPRIM_TRILIST
3DPRIM_TRISTRIP
3DPRIM_TRISTRIP_REVERSE

Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command. The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

8.3.3.2 3DSTATE_POLY_STIPPLE_OFFSET

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

The 3DSTATE_POLY_STIPPLE_OFFSET command is used to specify the origin of the repeated screen-space Polygon Stipple Pattern as an X,Y offset from the Color Buffer origin.

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 06h 3DSTATE_POLY_STIPPLE_OFFSET 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 Project: All
1	31:13	**Reserved** Project: All Format: MBZ
	12:8	**Polygon Stipple X Offset** Project: All Format: U5 FormatDesc Range [0,31] Specifies a 5 bit x address offset in the poly stipple pattern
	7:5	**Reserved** Project: All Format: MBZ
	4:0	**Polygon Stipple Y Offset** Project: All Format: U5 FormatDesc Range [0,31] Specifies a 5 bit y address offset in the poly stipple pattern

8.3.3.3 3DSTATE_POLY_STIPPLE_PATTERN

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

The 3DSTATE_POLY_STIPPLE_PATTERN command is used to specify the 32x32 Polygon Stipple Pattern used in the Polygon Stipple function of the WM unit.

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 07h 3DSTATE_POLY_STIPPLE_PATTERN Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 1Fh Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Polygon Stipple Pattern Row 1 (top most)** Project: All Format: 32 bit mask. Bit 31 = upper left corner, Bit 0 = upper right corner of first row. FormatDesc Specifies a pattern used by Polygon Stipple to mask out specific pixels of every 32x32 area rendered.
2..32	31:0	**Polygon Stipple Pattern Rows 2-32 (bottom most)** Project: All Format: 32 bit mask. Bit 31 = upper left corner, Bit 0 = upper right corner of first row. FormatDesc Specifies a pattern used by Polygon Stipple to mask out specific pixels of every 32x32 area rendered.

8.3.3.4 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP

3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP			
Project:	All	**Length Bias:**	2
The 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP command is used to specify the clamp used in the depth bias function of the WM unit.			

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 09h 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP 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 Project: All
1	31:0	**Global Depth Offset Clamp** Project: All Format: IEEE_FP FormatDesc This field specifies the *GlobalDepthOffsetClamp* term used in the Global Depth Offset Function

8.4 Early Depth/Stencil Processing

The Windower/IZ unit provides the Early Depth Test function, a major performance-optimization feature where an attempt is made to remove pixels that fail the Depth and Stencil Tests prior to pixel shading. This requires the WM unit to perform the interpolation of pixel ("source") depth values, read the current ("destination") depth values from the cached depth buffer, and perform the Depth and Stencil Tests. As the WM unit has per-pixel source and destination Z values, these values are passed in the PS thread payload, if required.

8.4.1 Depth Coefficient Read-Back

The WM unit must read back the depth coefficients from the URB entry containing the output of the Setup kernel. The value to program into the **Depth Coefficient URB Read Offset** state variable (in WM_STATE) should be computed as follows:

Depth Coefficient URB Read Offset = element_entry * 2 + 1

where element_entry is the location of the position element in the vertex data (ignoring the vertex header). For most applications, the position element will be in element 0.

8.4.2 Depth Offset

There are occasions where the Z position of some objects need to be slightly offset in order to reduce artifacts due to coplanar or near-coplanar primitives. A typical example is drawing the edges of triangles as wireframes – the lines need to be drawn slightly closer to the viewer to ensure they will not be occluded by the underlying polygon. Another example is drawing objects on a wall – without a bias on the z positions, they might be fully or partially occluded by the wall.

The device supports *global* depth offset, applied only to triangles, that bases the offset on the object's z slope. Note that there is no clamping applied at this stage after the Z position is offset – clamping to [0,1] can be performed later after the Z position is interpolated to the pixel. This is preferable to clamping prior to interpolation, as the clamping would change the Z slope of the entire object.

The Global Depth Offset function is controlled by the **Global Depth Offset Enable** state variable in WM_STATE. Global Depth Offset is only applied to 3DOBJ_TRIANGLE objects.

When Global Depth Offset Enable is ENABLED, the pipeline will compute:
MaxDepthSlope = max(abs(dZ/dX),abs(dz/dy)) // approximation of max depth slope for polygon
When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

```
Bias = GlobalDepthOffsetConstant * r +
       GlobalDepthOffsetScale * MaxDepthSlope
```

Where r is the minimum representable value > 0 in the depth buffer format, converted to float32. (note: If state bit **Legacy Global Depth Bias Enable** is set, the r term will be forced to 1.0)

When Floating Point Depth Buffer at Output Merger:

```
Bias = GlobalDepthOffsetConstant * 2^(exponent(max z in primitive)
        - r) + GlobalDepthOffsetScale * MaxDepthSlope
```

Where r is the # of mantissa bits in the floating point representation (excluding the hidden bit), e.g. 23 for float32. (note: If state bit Legacy Global Depth Bias Enable is set, no scaling is applied to the GobalDepthOffsetConstant).

Adding Bias to z:

```
if (GlobalDepthOffsetClamp > 0)
      Bias = min(DepthBiasClamp, Bias)
else if(GlobalDepthOffsetClamp < 0)
      Bias = max(DepthBiasClamp, Bias)
// else if GlobalDepthOffsetClamp == 0, no clamping occurs
z = z + Bias
```

Biasing is constant for a given primitive. The biasing formulas are performed with float32 arithmetic. Global Depth Bias is not applied to any point or line primitives.

8.4.3 Early Depth Test / Stencil Test / Write

When **Early Depth Test Enable** is ENABLED, the WM unit will attempt to discard depth-occluded pixels during scan conversion (before processing them in the Pixel Shader). Pixels are only discarded when the WM unit can ensure that they would have no impact to the ColorBuffer or DepthBuffer. This function is therefore only a performance feature.

If some pixels within a subspan are discarded, only the pixel mask is affected indicating that the discarded pixels are not active. If all pixels within a subspan are discarded, that subspan will not even be dispatched.

8.4.3.1 Software-Provided PS Kernel Info

In order for the WM unit to properly perform Early Depth Test and supply the proper information in the PS thread payload (and even determine if a PS thread needs to be dispatched), it requires information regarding the PS kernel operation. This information is provided by a number of state bits in WM_STATE, as summarized in the following table.

State Bit	Description
Pixel Shader Kill Pixel	This must be set when there is a chance that valid pixels passed to a PS thread may be discarded. This includes the discard of pixels by the PS thread resulting from a "killpixel" or "alphatest" function or as dictated by the results of the sampling of a "chroma-keyed" texture. The WM unit needs this information to prevent early depth/stencil writes for pixels which might be killed by the PS thread, etc. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Computed Depth	This must be set when the PS thread computes the "source" depth value (i.e., from the API POV, writes to the "oDepth" output). In this case the WM unit can't make any decisions based on the WM-interpolated depth value. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Uses Source Depth	Must be set if the PS thread requires the WM-interpolated source depth value. This will force the source depth to be passed in the thread payload where otherwise the WM unit would not have seen it as required. See WM_STATE/3DSTATE_WM for more information.

8.4.3.2 Early Depth Test Cases

There are cases, however, where the early depth test cannot be completed without information that will be generated by the pixel shader thread. The cases of depth test are divided as follows:

- **Computed depth (C)** is active whenever depth test *and* depth write (if enabled) needs to be performed post pixel shader. Most commonly, this includes cases where the pixel shader program writes to oDepth, emitting a "source depth" value which overrides the interpolated depth value. For these cases, the depth test cannot be done early, as the source depth is not available. Stencil test could be done early, but because the depth test cannot be done, the stencil write cannot be completed. Therefore, there is no advantage to doing the stencil test early. This includes cases where the pixel shader can kill pixels, including via sampler chroma key, as well as cases where the alpha test function is enabled, which kills pixels based on a programmable alpha test. In this case, even if the depth test fails, the pixel cannot be killed if a stencil write is indicated. Whether or not the stencil write happens depends on whether or not the pixel is killed later.
- **Non-promoted depth (N)** is active whenever the depth test can be done early but it cannot determine whether or not to write source depth to the depth buffer, therefore the depth write must be performed post pixel shader. This includes cases where the pixel shader can kill pixels, including via sampler chroma key, as well as cases where the alpha test function is enabled, which kills pixels based on a programmable alpha test. In this case, even if the depth test fails, the pixel cannot be killed if a stencil write is indicated. Whether or not the stencil write happens depends on whether or not the pixel is killed later. In these cases if stencil test fails and stencil writes are off, the pixels can also be killed early. If stencil writes are enabled, the pixels must be treated as Computed depth (described above).
- **Promoted depth (P)** is active whenever both the depth test and the conditional depth write can be performed before the pixel shader is executed. In this case, the entire depth/stencil operation is completed pre pixel shader. This includes all cases where depth test is disabled and stencil test is either disabled or no write is indicated.

The following logic equations define the test signals used by the following table. Also defined are the read enables that control reading of the depth/stencil buffer. Note that the **depth_test_en, stencil_test_en** and **depth_write_en** signals are qualified with a non-null depth buffer surface type (as specified in 3DSTATE_DEPTH_BUFFER).

```
depth_test_en = state_depth_test_en && !depth_surface_type_null

depth_read_en = depth_test_en

stencil_test_en = state_stencil_test_en && !depth_surface_type_null

stencil_read_en = state_stencil_test_en

depth_buffer_read_en = depth_read_en || stencil_read_en

depth_buffer_write_enable = state_depth_buffer_write_enable &&
      !depth_surface_type_null

stencil_buffer_write_en =  state_stencil_buffer_write_enable &&
      stencil_test_en
```

The following table indicates how the hardware determines which of the three above modes is active based on the above inputs. Note that cases where the stencil buffer write enable is active without the stencil test enable are not possible based on the equation above.

Clarify below text:

If statistics are enabled, windower (and Jitter) will need to detect when alpha test or killpix is on and the IZ Table output is Promoted (early depth test enabled or disabled). If these conditions are met, windower must force a write only depth allocation. In addition the windower / Jitter will force the result to be NONPROMOTED and force **Source Depth to Render Target** signal to be set. If **Pixel Shader Computed Depth** is not set, windower / Jitter must force the **Source Depth Present To EU** signal to be set and include the source depth data in the dispatch payload.

Behavior for Early Depth Test enabled:

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
0	0	0	0	0	0	P	0	0	0	0
0	0	0	0	0	1	P	0	0	0	0
0	0	0	0	1	0	P	0	0(1)[1]	0	0
0	0	0	0	1	1	P	0	0(1)[1]	0	0
0	0	0	1	0	0	P	0	0	0	0
0	0	0	1	0	1	N	1	1	0	0
0	0	0	1	1	0	N	0	1	0	0
0	0	0	1	1	1	N	0	1	0	0
0	0	1	0	0	0	P	0	0	0	0
0	0	1	0	0	1	P	0	0	0	0
0	0	1	0	1	0	C	0	1	1	0
0	0	1	0	1	1	C	0	1	1	0
0	0	1	1	0	0	P	0	0	0	0
0	0	1	1	0	1	N	1	1	0	0
0	0	1	1	1	0	C	0	1	1	0
0	0	1	1	1	1	C	0	1	1	0
1	0	0	0	0	0	P	0	0	0	0
1	0	0	0	0	1	P	0	0	0	0
1	0	0	0	1	0	P	0	0(1)[1]	0	0
1	0	0	0	1	1	P	0	0(1)[1]	0	0
1	0	0	1	0	0	P	0	0	0	0
1	0	0	1	0	1	N	1	1	0	1
1	0	0	1	1	0	N	0	1	0	1
1	0	0	1	1	1	N	0	1	0	1
1	0	1	0	0	0	P	0	0	0	0
1	0	1	0	0	1	P	0	0	0	0
1	0	1	0	1	0	C	0	1	1	1
1	0	1	0	1	1	C	0	1	1	1
1	0	1	1	0	0	P	0	0	0	0
1	0	1	1	0	1	N	1	1	0	1
1	0	1	1	1	0	C	0	1	1	1

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
1	0	1	1	1	1	C	0	1	1	1
1	1	0	0	0	0	P	0	0	0	0
1	1	0	0	0	1	C	0	0	0	1
1	1	0	0	1	0	P	0	0(1)[1]	0	0
1	1	0	0	1	1	C	0	1	0	1
1	1	0	1	0	0	P	0	0	0	0
1	1	0	1	0	1	C	1	1	0	1
1	1	0	1	1	0	C(N)[1]	0	1	0	1
1	1	0	1	1	1	C	0	1	0	1
1	1	1	0	0	0	P	0	0	0	0
1	1	1	0	0	1	C	1	1	1	1
1	1	1	0	1	0	C	0	1	1	1
1	1	1	0	1	1	C	0	1	1	1
1	1	1	1	0	0	P	0	0	0	0
1	1	1	1	0	1	C	1	1	1	1
1	1	1	1	1	0	C	0	1	1	1
1	1	1	1	1	1	C	0	1	1	1

NOTES:

1. The value in parenthesis is for [DevBW-A] only.

Behavior for Early Depth Test disabled:

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
0	0	0	0	0	0	P	0	0	0	0
0	0	0	0	0	1	P	0	0	0	0
0	0	0	0	1	0	C	0	1	0	0
0	0	0	0	1	1	C	0	1	0	0
0	0	0	1	0	0	C	1	1	0	0
0	0	0	1	0	1	C	1	1	0	0
0	0	0	1	1	0	C	0	1	0	0
0	0	0	1	1	1	C	0	1	0	0
0	0	1	0	0	0	C	1	1	1	0
0	0	1	0	0	1	C	1	1	1	0
0	0	1	0	1	0	C	0	1	1	0
0	0	1	0	1	1	C	0	1	1	0
0	0	1	1	0	0	C	1	1	1	0
0	0	1	1	0	1	C	1	1	1	0
0	0	1	1	1	0	C	0	1	1	0
0	0	1	1	1	1	C	0	1	1	0
1	0	0	0	0	0	C	0	0	0	1
1	0	0	0	0(1)[1]	1(0)[1]	C	0	0	0	1
1	0	0	0	1(0)[1]	0(1)[1]	C	0	1	0	1
1	0	0	0	1	1	C	0	1	0	1
1	0	0	1	0	0	C	1	1	0	1
1	0	0	1	0	1	C	1	1	0	1
1	0	0	1	1	0	C	0	1	0	1
1	0	0	1	1	1	C	0	1	0	1
1	0	1	0	0	0	C	1	1	1	1
1	0	1	0	0	1	C	1	1	1	1
1	0	1	0	1	0	C	0	1	1	1
1	0	1	0	1	1	C	0	1	1	1
1	0	1	1	0	0	C	1	1	1	1
1	0	1	1	0	1	C	1	1	1	1

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
1	0	1	1	1	0	C	0	1	1	1
1	0	1	1	1	1	C	0	1	1	1
1	1	0	0	0	0	C	0	0	0	1
1	1	0	0	0	1	C	0	0	0	1
1	1	0	0	1	0	C	0	1	0	1
1	1	0	0	1	1	C	0	1	0	1
1	1	0	1	0	0	C	1	1	0	1
1	1	0	1	0	1	C	1	1	0	1
1	1	0	1	1	0	C	0	1	0	1
1	1	0	1	1	1	C	0	1	0	1
1	1	1	0	0	0	C	1	1	1	1
1	1	1	0	0	1	C	1	1	1	1
1	1	1	0	1	0	C	0	1	1	1
1	1	1	0	1	1	C	0	1	1	1
1	1	1	1	0	0	C	1	1	1	1
1	1	1	1	0	1	C	1	1	1	1
1	1	1	1	1	0	C	0	1	1	1
1	1	1	1	1	1	C	0	1	1	1

NOTE:

1. The value in parenthesis is for [DevBW-A] only.

Note: Source depth present (to EU) will also be set in cases in which the pixel shader uses source depth (vPos.z) regardless of any other condition.

The specific actions for each case are as follows.

Early Depth Mode	Pixel	Depth	Stencil	Depth sent to Pixel Shader	Depth sent to Render Target	Stencil sent to PS/RT
Computed Depth	conditionally killed based on depth/stencil test post-shader	tested and written post-shader	tested and written post-shader	source depth for vPos.z if used dest depth passed through	source depth from oDepth dest depth passed through	dest stencil passed through if stencil test enabled
Non-promoted	pixel killed pre-shader if depth test fails and no stencil write indicated	test pre-and post-shader, written post-shader	tested and written post-shader	source depth for vPos.z if used source depth always	source depth from vPos.z	dest stencil passed through if stencil test enabled
Promoted	pixel killed pre-shader on fail	tested and written pre-shader	tested and written pre-shader	source depth for vPos.z if used	none	none

The following psuedocode describes the logic that determines whether color, depth, and stencil are written depending on results of alpha, depth, and stencil tests.

```
alpha_test_pass = TRUE
depth_test_pass = TRUE
stencil_test_pass = TRUE

if (alpha_test_enable) alpha_test_pass = TestAlpha();
if (depth_test_enable) depth_test_pass = TestDepth();
if (stencil_test_enable) stencil_test_pass = TestStencil();

stencil_update = (new_stencil_value != dst_stencil_value) &&
       (stencil_test_enable == TRUE)

pass_color_depth = (alpha_test_pass == TRUE) && (depth_test_pass == TRUE)
       && (stencil_test_pass == TRUE) && (pixel_enabled == TRUE)
pass_stencil = (alpha_test_pass == TRUE) && (stencil_update == TRUE) &&
       (pixel_enabled == TRUE)

pixel_color_write = pass_color_depth && (color_component_write_disables
       != 0xf)
pixel_depth_write = pass_color_depth && (depth_buffer_write_enable ==
       TRUE)
pixel_stencil_write = pass_stencil && (stencil_buffer_write_enable ==
       TRUE)
```

8.4.4 Depth/Stencil Buffer State

8.4.4.1 3DSTATE_DEPTH_BUFFER

3DSTATE_DEPTH_BUFFER			
Project:	All	**Length Bias:**	2
The depth buffer surface state is delivered as a non-pipelined state packet.			

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 05h 3DSTATE_DEPTH_BUFFER Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 3h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All

3DSTATE_DEPTH_BUFFER

1	31:29	**Surface Type** Project: All Format: U3 Enumerated Type This field defines the type of the surface. **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 / All 4h-6h / Reserved / / All 7h / SURFTYPE_NULL / Defines a null surface / All **Programming Notes** The **Surface Type** of the depth buffer must be the same as the **Surface Type** of the render target(s) (defined in SURFACE_STATE), unless either the depth buffer or render targets are SURFTYPE_NULL.
	28	**Reserved** Project: All Format: MBZ
	27	**Tiled Surface** Project: All Format: U1 enumerated type FormatDesc Specifies if the surface is tiled. **Value** / **Name** / **Description** / **Project** 0h / TRUE / Tiled / All 1h / FALSE / Linear / All **Programming Notes** / **Project** Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory. / All The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit. / All **Errata** / **Description** / **Project** BWT014 / The Depth Buffer Must be Tiled, it cannot be linear. This field must be set to 1 on DevBW-A. / [DevBW-A,B

3DSTATE_DEPTH_BUFFER

	26	**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. The Depth Buffer, if tiled, must use Y-Major tiling. See *Memory Interface Functions* for details on memory tiling and restrictions. This field is ignored when the surface is linear. **Value** **Name** **Description** **Project** 0h Reserved All 1h TILEWALK_YMAJOR Y major tiled All
	25	**Depth Buffer Coordinate Offset Disable** Project: All Format: Disable FormatDesc Disables the application (addition) of the “upper bits” of the Drawing Rectangle Origin to Depth Buffer coordinates. (This does not affect the application of the Drawing Rectangle Origin to the Color Buffer coordinates). This control is provided to better support “Front Buffer Rendering”. By disabling the Draw Rectangle adjustment of Depth Buffer coordinates, software can utilize a “window-sized” Depth Buffer while rendering to a window within the Color Buffer. Without this control, use of the Draw Rectangle adjustment would require the Depth Buffer to be dimensioned to match the Color Buffer (screen) vs. the target window. **Programming Notes** **Project** The device still applies some small coordinate offset in order to provide the required alignment of color and depth memory/cache accesses. Software needs to consider this alignment when allocating depth buffers. All This bit must not be set when rendering to field-mode (interlaced) Color Buffers (i.e., when Surface State’s VerticalLineStride==1). All This bit can only be set when rendering to surfaces of type SURFTYPE_1D and SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped) All

3DSTATE_DEPTH_BUFFER

	24:23	**Software Tiled Rendering Mode** Project: All Format: U2 enumerated type FormatDesc This field is intended to enable *software tiled rendering (STR)*. If certain restrictions are met, performance can be improved by reducing memory bandwidth to the render target and depth buffer. **Normal mode:** Rendering behaves normally. **STR1 mode:** Only pixels within a particular 64x32 block (aligned relative to the upper left corner of the render target) are rendered between pixel shader serializations. Generally the alignment is guaranteed via a scissor rectangle. A write to a given pixel in the render target must occur before a read from the same pixel. **STR2 mode:** The restrictions of STR1 mode applies, and in addition each pixel must be rendered with depth write enabled and depth test disabled before it can be rendered with depth test enabled. The depth buffer in memory is not updated, even on a render cache flush. Depth buffer data is contained only within the render cache during rendering.
	22	Reserved: MBZ
	21	Reserved: MBZ

Value	Name	Description	Project
0h	NORMAL	Normal mode	All
1h	STR1	STR1 mode	Reserved
2h	Reserved		All
3h	STR2	STR2 mode	Reserved

Programming Notes	Project
only normal mode is supported	All
The render cache must be flushed when this field is modified from its previous state	All
For both STR modes, the depth buffer (if used) must be tiled Y with D16_UNORM format, and the render target surface must be tiled X or Y	All
For both STR modes, the only data port messages allowed that use the render cache are the Render Target UNORM Read and Write messages.	All
Performance considerations: Both STR modes eliminate all memory read traffic from the render target. The STR2 mode additionally eliminates all memory traffic to the depth buffer.	All
When STR2 mode is used in conjunction with the advanced scheduler, context switches can only occur on the boundaries between the 64x32 blocks, as the depth buffer contents are not saved for restore when the context is restarted.	All

3DSTATE_DEPTH_BUFFER

	20:18	**Surface Format** Project: All Format: U3 enumerated type FormatDesc Specifies the format of the depth buffer. **Value** **Name** **Description** **Project** 0h D32_FLOAT_S8X24_UINT All 1h D32_FLOAT All 2h D24_UNORM_S8_UINT All 3h D24_UNORM_X8_UINT Reserved 4h Reserved All 5h D16_UNORM All 6h-7h Reserved All
	17	**Reserved** Project: All Format: MBZ
	16:0	**Surface Pitch** Project: All Format: U17 pitch in (Bytes – 1) FormatDesc Range if linear: [63, 128K-1] corresponding to [64B, 128KB] also restricted to a multiple of 64B if tiled: [127, 128K-1] corresponding to [128B, 128KB] also restricted to a multiple of 128B This field specifies the pitch of the depth buffer in (#Bytes – 1). **Programming Notes** **Project** If this surface is tiled, the pitch specified must be a multiple of the tile pitch, in the range [128B, 128KB]. All If the surface is linear, the pitch can be any multiple of 64 bytes up to 128KB. All
2	31:0	**Surface Base Address** Project: All Address: GraphicsAddress[31:0] This field specifies the starting DWord address of the buffer in mapped Graphics Memory. **Programming Notes** The Depth Buffer can only be mapped to Main Memory (uncached). If the surface is tiled, the base address must conform to the Per-Surface Tiling Alignment Rules as documented in TBD. If the buffer is linear, the surface must be 64-byte aligned.

3DSTATE_DEPTH_BUFFER

3	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] This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. **Programming Notes** The **Height** of the depth buffer must be the same as the **Height** of the render target(s) (defined in SURFACE_STATE), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped).
	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] 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. **Programming Notes** — **Project** The Width specified by this field must be less than or equal to the surface pitch (specified in bytes via the **Surface Pitch** field). — All For cube maps, **Width** must be set equal to **Height**. — All The **Width** of the depth buffer must be the same as the **Width** of the render target(s) (defined in SURFACE_STATE), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped). — All
	5:2	**LOD** Project: All Format: U4 in LOD units FormatDesc Range [0, 13] This field defines the MIP level that is currently being rendered into. **Programming Notes** — **Project** The **LOD** of the depth buffer must be the same as the **LOD** of the render target(s) (defined in SURFACE_STATE). — All

	1	**MIP Map Layout Mode** Project: All Format: U1 enumerated type FormatDesc **For 1D and 2D Surfaces:** 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 **Value Name Description Project** 0h MIPLAYOUT_BELOW All 1h MIPLAYOUT_RIGHT All **Programming Notes Project** MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces All
	0	**Reserved** Project: All Format : MBZ
4	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 (r/z dimension) [0,2047] SURFTYPE_CUBE: must be zero 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. **Programming Notes** The **Depth** of the depth buffer must be the same as the **Depth** of the render target(s) (defined in SURFACE_STATE).

3DSTATE_DEPTH_BUFFER		
	20:10	**Minimum Array Element** Project: All Format: U11 FormatDesc Range SURFTYPE_1D/2D: [0,511] SURFTYPE_3D: [0,2047] **For 1D and 2D Surfaces:** This field indicates the minimum array element that can be accessed as part of this surface. The delivered array index is added to this field before being used to address the surface. **For 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 is ignored. **Programming Notes** **Project** **[DevBW-A]:** this field must be zero. [DevBW-A]
	9:1	**Render Target View Extent** Project: All Format: U9 FormatDesc Range SURFTYPE_1D/2D: same value as Depth field SURFTYPE_3D: [0,511] to indicate extent of [1,512] **For 3D Surfaces:** This field indicates the extent of the accessible 'R' coordinates minus 1 on the LOD currently being rendered to. **For 1D and 2D Surfaces:** This field must be set to the same value as the **Depth** field. **For Other Surfaces:** This field is ignored. **Programming Notes** **Project** **[DevBW-A]:** this field must be zero [DevBW-A]
	0	**Reserved** Project: All Format: MBZ

8.5 Pixel Shader Thread Generation

After a group of object pixels have been rasterized, the Pixel Shader function is invoked to further compute pixel color/depth information and cause results to be written to rendertargets and/or depth buffers. For each pixel, the Pixel Shader calculates the values of the various vertex attributes that are to be interpolated across the object using the interpolation coefficients. It then executes an API-supplied Pixel Shader Program. Instructions in this program permit the accessing of texture map data, where Texture Samplers are employed to sample and filter texture maps (see the *Shared Functions* chapter). Arithmetic operations can be performed on the texture data, input pixel information and Pixel Shader Constants in order to compute the resultant pixel color/depth. The Pixel Shader program also allows the pixel to be discarded from further processing. For pixels that are not discarded, the pixel shader must send messages to update one or more render targets with the pixel results.

8.5.1 Pixel Grouping (Dispatch Size) Control

The WM unit can pass a grouping of 2 subspans (8 pixels), 4 subspans (16 pixels) or 8 subspans (32 pixels) to a Pixel Shader thread. Software should take into account the following considerations when determining which groupings to support/enable during operation. This determination involves a tradeoff of these likely conflicting issues. Note that the size of the dispatch has significant impact on the kernel program (it is certainly not transparent to the kernel). Also note that there is no implied spatial relationship between the subspans passed to a PS thread, other than the fact that they come from the same object.

1. **Thread Efficiency**: In general, there is some amount of overhead involved with PS thread dispatch, and if this can be amortized over a larger number of pixels, efficiency will likely increase. This is especially true for very short PS kernels, as may be used for desktop composition, etc.
2. **GRF Consumption**: Processing more pixels per thread will require a larger thread payload and likely more temporary register usage, both of which translate into a requirement for a larger GRF register allocation for the threads. If this increased GRF usage could lead to increased use of scratch space (for spill/fill, etc.) and possibly less efficient use of the EUs (as it would be less likely to find an EU with enough free physical GRF registers to service the thread).
3. **Object Size**: If the number of very small objects (e.g., covering 2 subspans or fewer) is expected to comprise a significant portion of the workload, supporting the 8-pixel dispatch mode may be advantageous. Otherwise there could be a large number of 16-pixel dispatches with only 1 or 2 valid subspans, resulting in low efficiency for those threads.
4. **Intangibles**: Kernel footprint & Instruction Cache impact; Complexity; ….

The groupings of subspans that the WM unit is allowed to include in a PS thread payload are controlled by the **32,16,8 Pixel Dispatch Enable** state variables programmed in WM_STATE. Using these state variables, the WM unit will attempt to dispatch the largest allowed grouping of subspans. The following table lists the possible combinations of these state variables.

Note: In the following table, the Valid column indicates which products that combination is supported on. Combinations of dispatch enables not listed in the table are not available on any product.

A: Valid on all products

There is only one kernel start pointer (KSP) specified in WM_STATE, with other kernels being entered via an offset from the single KSP as follows:

KSP[0] = KSP

KSP[1] = KSP+1

KSP[2] = KSP+2

KSP[3] = KSP+3

All kernels share the same GRF register count field, with the one with the maximum register count required applying to all.

Table 8-1. Variable Pixel Dispatch

Contiguous 64 Pixel Dispatch Enable	Contiguous 32 Pixel Dispatch Enable	32 Pixel Dispatch Enable	16 Pixel Dispatch Enable	8 Pixel Dispatch Enable	Valid	IP for n-pixel Contiguous Dispatch		IP for n-pixel Dispatch (KSP offsets are in 128-bit instruction units)		
						n=64	n=32	n=32	n=16	n=8
0	0	0	0	1	A					KSP[0]
0	0	0	1	0	A				KSP[0]	
0	0	0	1	1	A				KSP[2]	KSP[0]
0	0	1	0	0	B			KSP[0]		
0	0	1	1	0	A			KSP[1]	KSP[2]	
0	0	1	1	1	A			KSP[1]	KSP[2]	KSP[0]
0	1	0	0	0	B		KSP[0]			
0	1	1	0	0	B		KSP[1]	KSP[0]		
0	1	1	1	0	B		KSP[2]	KSP[1]	KSP[0]	
1	0	0	0	0	B	KSP[0]				
1	0	1	0	0	B	KSP[1]		KSP[0]		
1	0	1	1	0	B	KSP[2]		KSP[1]	KSP[0]	
1	1	0	0	0	B	KSP[1]	KSP[0]			
1	1	1	0	0	B	KSP[2]	KSP[1]	KSP[0]		
1	1	1	1	0	B	KSP[3]	KSP[2]	KSP[1]	KSP[0]	

The WM unit will select the optimal dispatch size given the enabled modes and the number of subspans remaining in the object (n), via the following algorithm: (note: This algorithm assumes a valid set of state variables, as listed in Table 8-1).

```
if (32PixelDispatchEnable && n>7)
Dispatch 32 Pixels
else if (16PixelDispatchEnable && (n>2 || !8PixelDispatchEnable))
Dispatch 16 Pixels
else
Dispatch 8 Pixels
```

Depending on the subspan grouping selected, the WM unit will modify the starting PS Instruction Pointer (derived from the Kernel Start Pointer in WM_STATE) as a means to inform the PS kernel of the number of subspans included in the payload. The modified IP is a function of the enabled modes and the dispatch size, as shown in Table 8-1. The driver must ensure that the PS kernel begins with a corresponding jump table to properly handle the number of subspans dispatched. The WM unit will “OR” in the two lsbs of the Kernel Pointer (bits 5:4) to create an instruction level address (note that the pointer from WM_STATE is 64 byte aligned which correpsonds to four instructions).

If only one dispatch mode is enabled, the Jitter should not include any jump table entries at the beginning of the PS kernel. If multiple dispatch modes are enabled, a two entry jump table should always be inserted, regardless of which modes are enabled (jump table entry for 8 pixel dispatch, followed by jump table entry for 32 pixel dispatch).

Note that for a 32 pixel dispatch, the Windower will mulitply the **Dispatch GRF Start Register for URB Data** state by 2 to account for the extra payload data required. The Pixel Shader kernel needs to comprehend this modification for the 32 pixel kernel code.

8.5.2 PS Thread Payload for Normal Dispatch

The following table lists all possible contents included in a PS thread payload, in the order they are provided. Certain portions of the payload are optional, in which case the corresponding phase is skipped.

All registers are numbered starting at 0, but many registers are skipped depending on configuration. This causes all registers below to be renumbered to fill in the skipped locations. The only case where actual registers may be skipped is immediately before the CURBE data and again before the setup URB data.

DWord	Bit	Description
R0.7	31	**Snapshot Flag:** If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:24	Reserved
	23:0	**Primitive Thread ID:** This field contains the primitive thread count passed to the Windower from the Strips Fans Unit. (See *Debug* for further description). Format: Reserved for HW Implementation Use.
R0.6	31:24	Reserved
	23:0	**Thread ID:** This field contains the thread count which is incremented by the Windower for every thread that is dispatched. (See *Debug* for further description). Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer:** Specifies the 1K-byte aligned pointer to the scratch space available for this PS thread. This is specified as an offset to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	**Color Code:** This ID is assigned by the Windower unit and is used to track synchronizng events. Format: Reserved for HW Implementation Use.
	7:0	**FFTID:** This ID is assigned by the WM unit and is a identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for HW Implementation Use.

DWord	Bit	Description
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the Sampler State table. It is specified as an offset from the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. **Programming Notes:** • **[DevBW-A] A0 Erratum BWT005**: The range [0,11] for this register indicates [1KB, 12KB] in 1K byte increments. If MMIO register 21D0h bit 3 is set, then value 11 is an exception and indicates a 256KB space instead of 12KB. Note that Scratch Space Base Pointer must be 8MB-aligned in order to set the 256KB scratch space. • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:0	Reserved : delivered as zeros (reserved for message header fields)
R0.1	31:6	**Color Calculator State Pointer:**Specifies the 64-byte aligned pointer to the Color Calculator state (CC_STATE structure in memory). It is specified as an offset from the **General State Base Address**. This value is eventually passed to the ColorCalc function in the DataPort and is used to fetch the corresponding CC_STATE data. Format = GeneralStateOffset[31:5]
	5:0	Reserved
R0.0	31:16	**Pixel Mask (SubSpan[3:0]) :** Indicates which pixels within the four subspans are lit. If 32 pixel dispatch is enabled, this field contains the pixel mask for the first four subspans. Note: This is not a duplicate of the Dispatch Mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel Mask Copy (SubSpan[3:0]) :** This is a duplicate copy of the pixel mask. This copy can be modified as the pixel shader thread executesin order to turn off pixels based on kill instructions.
R1.7	31	Reserved
	30:27	**Viewport Index:** Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td>26:16</td><td>Render Target Array Index:Specifies the array index to be used for the following surface types:
SURFTYPE_1D: specifies the array index. Range = [0,511]
SURFTYPE_2D: specifies the array index. Range = [0,511]
SURFTYPE_3D: specifies the “r” coordinate. Range = [0,2047]
SURFTYPE_CUBE: specifies the face identifier. Range = [0,5]
<table>
<tr><th>face</th><th>Render Target Array Index</th></tr>
<tr><td>+x</td><td>0</td></tr>
<tr><td>-x</td><td>1</td></tr>
<tr><td>+y</td><td>2</td></tr>
<tr><td>-y</td><td>3</td></tr>
<tr><td>+z</td><td>4</td></tr>
<tr><td>-z</td><td>5</td></tr>
</table>
Format = U11</td></tr>
<tr><td></td><td>15:0</td><td>Reserved</td></tr>
<tr><td>R1.6</td><td>31</td><td>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</td></tr>
<tr><td></td><td>30</td><td>Source Depth Present:Indicates that source depth is included in the dispatch</td></tr>
<tr><td></td><td>29</td><td>Source Depth to Render Target: Indicates that source depth will be sent to the render target</td></tr>
<tr><td></td><td>28</td><td>Destination Depth Present: Indicates that destination depth is included in the dispatch and sent to the render target</td></tr>
<tr><td></td><td>27</td><td>Destination Stencil Present: Indicates that destination stencil is included in the dispatch and sent to the render target</td></tr>
<tr><td></td><td>26</td><td>Antialias Alpha to Render Target: Indicates to the PS thread that antialias alpha data must be included in render target writes (i.e., included in the DataPort RT Write message payload). The WM unit generates this control bit based on object type and state settings. This indication is required as the PS kernel is likely shared between anti-aliased and non-anti-aliased objects.
This bit applies to all subspans (i.e., both sets of 4 subspans for 32-pixel dispatches).
By definition, Antialias Alpha Present will also be set.
Format: Enable</td></tr>
<tr><td></td><td>25</td><td>Antialias Alpha Present: Indicates that antialias alpha data is included in this PS thread payload.
This bit applies to all subspans (i.e., both sets of 4 subspans for 32-pixel dispatches).
Format: Enable</td></tr>
<tr><td></td><td>24:5</td><td>Reserved</td></tr>
<tr><td></td><td>4:0</td><td>Primitive Topology Type:This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST).
Format: (See 3DPRIMITIVE command in 3D Pipeline)</td></tr>
</table>

DWord	Bit	Description
R1.5	31:16	**Y3:** Y coordinate (screen space) for upper-left pixel of subspan 3 Format = U16
	15:0	**X3:** X coordinate (screen space) for upper-left pixel of subspan 3 Format = U16
R1.4	31:16	**Y2 :** Y coordinate (screen space) for upper-left pixel of subspan 2 Format = U16
	15:0	**X2 :** X coordinate (screen space) for upper-left pixel of subspan 2 Format = U16
R1.3	31:16	**Y1 :** Y coordinate (screen space) for upper-left pixel of subspan 1 Format = U16
	15:0	**X1 :** X coordinate (screen space) for upper-left pixel of subspan 1 Format = U16
R1.2	31:16	**Y0 :** Y coordinate (screen space) for upper-left pixel of subspan 0 Format = U16
	15:0	**X0 :** X coordinate (screen space) for upper-left pixel of subspan 0 Format = U16
R1.1	31:0	**Ystart :** Y coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float
R1.0	31:0	**Xstart:** X coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float
		The following data is optional depending on the state relating to depth / stencil / alpha present flags above. Phases including only data for subspans 2 and 3 *are* included for 8-pixel dispatches, even though they do not contain valid data. Following the optional data is the attribute interpolation coefficient data
		R2-R3: delivered only if **Source Depth Present** is set.
R2.7	31:0	Interpolated Depth for Subspan 1, Pixel 3 (lower right) Format = IEEE_Float
R2.6	31:0	Interpolated Depth for Subspan 1, Pixel 2 (lower left)
R2.5	31:0	Interpolated Depth for Subspan 1, Pixel 1 (upper right)
R2.4	31:0	Interpolated Depth for Subspan 1, Pixel 0 (upper left)
R2.3	31:0	Interpolated Depth for Subspan 0, Pixel 3 (lower right)
R2.2	31:0	Interpolated Depth for Subspan 0, Pixel 2 (lower left)
R2.1	31:0	Interpolated Depth for Subspan 0, Pixel 1 (upper right)
R2.0	31:0	Interpolated Depth for Subspan 0, Pixel 0 (upper left)
R3.7	31:0	Interpolated Depth for Subspan 3, Pixel 3 (lower right)
R3.6	31:0	Interpolated Depth for Subspan 3, Pixel 2 (lower left)
R3.5	31:0	Interpolated Depth for Subspan 3, Pixel 1 (upper right)
R3.4	31:0	Interpolated Depth for Subspan 3, Pixel 0 (upper left)

DWord	Bit	Description
R3.3	31:0	Interpolated Depth for Subspan 2, Pixel 3 (lower right)
R3.2	31:0	Interpolated Depth for Subspan 2, Pixel 2 (lower left)
R3.1	31:0	Interpolated Depth for Subspan 2, Pixel 1 (upper right)
R3.0	31:0	Interpolated Depth for Subspan 2, Pixel 0 (upper left)
		R4: delivered only if **Antialias Alpha Present** or **Destination Stencil Present** is set. The Antialias Alpha data is only valid if **Antialias Alpha Present** is set, and likewise the Destination Stencil data is only valid if **Destination Stencil Present** is set.
		[DevBW, DevCL]
R4.7	31:28	Antialias Alpha for Subspan 3, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.4
	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)
R4.6	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)
	3:0	Antialias Alpha for Subspan 0, Pixel 0 (upper left)
R4.5:4		Reserved
R4.3	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)
R4.2	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)
R4.1	31:24	Destination Stencil for Subspan 1, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 1, Pixel 2 (lower left)

DWord	Bit	Description
	15:8	Destination Stencil for Subspan 1, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 1, Pixel 0 (upper left)
R4.0	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)
		R5-R6: delivered only if **Destination Depth Present** is set.
R5.7	31:0	Destination Depth for Subspan 1, Pixel 3 (lower right) Format depends on depth buffer surface format, and is intended to be passed through to the render target without modification by software.
R5.6	31:0	Destination Depth for Subspan 1, Pixel 2 (lower left)
R5.5	31:0	Destination Depth for Subspan 1, Pixel 1 (upper right)
R5.4	31:0	Destination Depth for Subspan 1, Pixel 0 (upper left)
R5.3	31:0	Destination Depth for Subspan 0, Pixel 3 (lower right)
R5.2	31:0	Destination Depth for Subspan 0, Pixel 2 (lower left)
R5.1	31:0	Destination Depth for Subspan 0, Pixel 1 (upper right)
R5.0	31:0	Destination Depth for Subspan 0, Pixel 0 (upper left)
R6.7	31:0	Destination Depth for Subspan 3, Pixel 3 (lower right)
R6.6	31:0	Destination Depth for Subspan 3, Pixel 2 (lower left)
R6.5	31:0	Destination Depth for Subspan 3, Pixel 1 (upper right)
R6.4	31:0	Destination Depth for Subspan 3, Pixel 0 (upper left)
R6.3	31:0	Destination Depth for Subspan 2, Pixel 3 (lower right)
R6.2	31:0	Destination Depth for Subspan 2, Pixel 2 (lower left)
R6.1	31:0	Destination Depth for Subspan 2, Pixel 1 (upper right)
R6.0	31:0	Destination Depth for Subspan 2, Pixel 0 (upper left)
		R7: delivered only if this is a *32-pixel dispatch*.
R7.7	31:0	Reserved
R7.6	31:0	Reserved
R7.5	31:0	Reserved
R7.5	31:16	**Y7:** Y coordinate (screen space) for upper-left pixel of subspan 7 Format = U16
	15:0	**X7:** X coordinate (screen space) for upper-left pixel of subspan 7 Format = U16
R7.4	31:16	**Y6**
	15:0	**X6**
R7.3	31:16	**Y5**
	15:0	**X5**
R7.2	31:16	**Y4**
	15:0	**X4**

DWord	Bit	Description
R7.1	31:0	Reserved
R7.0	31:16	**Pixel Mask (SubSpan[7:4]) :** Indicates which pixels within the upper four subspans are lit. This field is valid only when the 32 pixel dispatch state is enabled. This field must not be modified by the pixel shader thread. Note: This is not a duplicate of the dispatch mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel Mask Copy (SubSpan[7:4]) :** This is a duplicate copy of pixel mask for the upper 16 pixels. This copy will be modified as the pixel shader thread executes to turn off pixels based on kill instructions.
		R8-R9: delivered only if **Source Depth Present** is set and this is a *32-pixel dispatch*.
R8.7	31:0	Interpolated Depth for Subspan 5, Pixel 3 (lower right) Format = IEEE_Float
R8.6	31:0	Interpolated Depth for Subspan 5, Pixel 2 (lower left)
R8.5	31:0	Interpolated Depth for Subspan 5, Pixel 1 (upper right)
R8.4	31:0	Interpolated Depth for Subspan 5, Pixel 0 (upper left)
R8.3	31:0	Interpolated Depth for Subspan 4, Pixel 3 (lower right)
R8.2	31:0	Interpolated Depth for Subspan 4, Pixel 2 (lower left)
R8.1	31:0	Interpolated Depth for Subspan 4, Pixel 1 (upper right)
R8.0	31:0	Interpolated Depth for Subspan 4, Pixel 0 (upper left)
R9.7	31:0	Interpolated Depth for Subspan 7, Pixel 3 (lower right)
R9.6	31:0	Interpolated Depth for Subspan 7, Pixel 2 (lower left)
R9.5	31:0	Interpolated Depth for Subspan 7, Pixel 1 (upper right)
R9.4	31:0	Interpolated Depth for Subspan 7, Pixel 0 (upper left)
R9.3	31:0	Interpolated Depth for Subspan 6, Pixel 3 (lower right)
R9.2	31:0	Interpolated Depth for Subspan 6, Pixel 2 (lower left)
R9.1	31:0	Interpolated Depth for Subspan 6, Pixel 1 (upper right)
R9.0	31:0	Interpolated Depth for Subspan 6, Pixel 0 (upper left)
		R10: delivered only if **Antialias Alpha Present** or **Destination Stencil Present** is set and this is a 32-pixel dispatch. The Antialias Alpha data is only valid if **Antialias Alpha Present** is set, and likewise the Destination Stencil data is only valid if **Destination Stencil Present** is set.
		[DevBW, DevCL]
R10.7	31:28	Antialias Alpha for Subspan 7, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.4
	27:24	Antialias Alpha for Subspan 7, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 7, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 7, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 6, Pixel 3 (lower right)

DWord	Bit	Description
	11:8	Antialias Alpha for Subspan 6, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 6, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 6, Pixel 0 (upper left)
R10.6	31:28	Antialias Alpha for Subspan 5, Pixel 3 (lower right)
	27:24	Antialias Alpha for Subspan 5, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 5, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 5, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 4, Pixel 3 (lower right)
	11:8	Antialias Alpha for Subspan 4, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 4, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 4, Pixel 0 (upper left)
R10.5:4		Reserved
R10.3	31:24	**Destination Stencil for Subspan 7, Pixel 3 (lower right) :** This field contains the destination stencil value associated with Pixel 3 of Subspan 7. Format = U8
	23:16	Destination Stencil for Subspan 7, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 7, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 7, Pixel 0 (upper left)
R10.2	31:24	Destination Stencil for Subspan 6, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 6, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 6, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 6, Pixel 0 (upper left)
R10.1	31:24	Destination Stencil for Subspan 5, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 5, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 5, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 5, Pixel 0 (upper left)
R10.0	31:24	Destination Stencil for Subspan 4, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 4, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 4, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 4, Pixel 0 (upper left)
		R11-R12: delivered only if **Destination Depth Present** is set and this is a *32-pixel dispatch*.
R11.7	31:0	Destination Depth for Subspan 5, Pixel 3 (lower right) Format = IEEE_Float
R11.6	31:0	Destination Depth for Subspan 5, Pixel 2 (lower left)
R11.5	31:0	Destination Depth for Subspan 5, Pixel 1 (upper right)
R11.4	31:0	Destination Depth for Subspan 5, Pixel 0 (upper left)
R11.3	31:0	Destination Depth for Subspan 4, Pixel 3 (lower right)
R11.2	31:0	Destination Depth for Subspan 4, Pixel 2 (lower left)

DWord	Bit	Description
R11.1	31:0	Destination Depth for Subspan 4, Pixel 1 (upper right)
R11.0	31:0	Destination Depth for Subspan 4, Pixel 0 (upper left)
R12.7	31:0	Destination Depth for Subspan 7, Pixel 3 (lower right)
R12.6	31:0	Destination Depth for Subspan 7, Pixel 2 (lower left)
R12.5	31:0	Destination Depth for Subspan 7, Pixel 1 (upper right)
R12.4	31:0	Destination Depth for Subspan 7, Pixel 0 (upper left)
R12.3	31:0	Destination Depth for Subspan 6, Pixel 3 (lower right)
R12.2	31:0	Destination Depth for Subspan 6, Pixel 2 (lower left)
R12.1	31:0	Destination Depth for Subspan 6, Pixel 1 (upper right)
R12.0	31:0	Destination Depth for Subspan 6, Pixel 0 (upper left)
		Optional Padding before the Start of URB-Sourced Data The locations between the end of the Optional Payload Header and the location programmed via **Dispatch GRF Start Register for URB Data** (if any) are considered "padding" and Reserved. (see below)
optional, multiple of 8 DWs	31:0	Reserved
		URB DATA STARTS HERE The **Dispatch GRF Start Register for URB Data** state variable in WM_STATE is used to define the starting location of URB-sourced data within the PS thread payload. This control is provided to allow the URB-sourced data to be located at a fixed location within thread payloads, regardless of the amount of data in the Optional Payload Header. This permits the kernel to use direct GRF addressing to access the URB-sourced data, regardless of the optional parameters being passed (as these are determined on-the-fly by the WM unit).
		Constant URB Entry (CURBE) Data Optionally, some amount of data (multiples of 8 DWs) can be read from the CURBE URB entry and placed in the thread payload at this point (after the variable payload header and prior to the Setup URB data). The amount of CURBE data provided is specified by **Constant URB Entry Read Length** in WM_STATE, and the starting read offset in that URB entry is specified by **Constant URB Entry Read Offset** in WM_STATE.
optional, multiple of 8 DWs	31:0	Constant Data
		Setup URB Data (Attribute Interpolation Coeffcients) Some amount of data (multiples of 8 DWs) can be read from the Setup URB entry and placed in the thread payload at this point (after the variable payload header and any CURBE data – i.e., the end of the payload). This data is read from the Setup URB entry based on the URB Handle associated with the object being rendered (as received from the SF unit). The amount of Setup URB data provided is specified by **Setup URB Entry Read Length** in WM_STATE, and the starting read offset in that URB entry is specified by **Setup URB Entry Read Offset** in WM_STATE. The order/content/format of this data is actually determined by the Setup kernel which is executed from the Strips Fans Unit. The following DWords are labelled assuming the typical/expected definition.

DWord	Bit	Description
Rp.7	31:0	Co[1] – Co Coefficient for Attribute [1] (optional)
Rp.6	31:0	Reserved
Rp.5	31:0	Cy[1] – Cy Coefficient for Attribute [1] (optional)
Rp.4	31:0	Cx[1] – Cx Coefficient for Attribute [1] (optional)
Rp.3	31:0	Co[0] – Co Coefficient for Attribute [0]
Rp.2	31:0	Reserved
Rp.1	31:0	Cy[0] – Cy Coefficient for Attribute [0]
Rp.0	31:0	Cx[0] – Cx Coefficient for Attribute [0]
R(p+1):Rq		Coefficients for additional attributes (optional) See definition of Rp for formats.

8.6 Other WM Functions

8.6.1 Statistics Gathering

If **Statistics Enable** is set in WM_STATE, the Windower increments the PS_INVOCATIONS_COUNT register once for each unmasked pixel that is *dispatched* to a Pixel Shader thread. If **Early Depth Test Enable** is set it is possible for pixels to be discarded prior to reaching the Pixel Shader due to failing the depth or stencil test. PS_INVOCATIONS_COUNT will still be incremented for these pixels since the depth test occurs after the pixel shader from the point of view of SW.

[DevBW] A0 Erratum BWT004 states that there is no way to indicate a true “null” pixel shader (in the sense that the pixel shader dispatch will be skipped.) The “dummy” PS thread required for a “null” pixel shader will still cause PS_INVOCATIONS_COUNT to increment on pixel dispatches; if the “null” pixel dispatches are not to be counted (D3D10 expects them not to be counted), **Statistics Enable** must be *cleared* when changing to a “null” pixel shader. Clearing **Statistics Enable** may also prevent PS_DEPTH_COUNT from incrementing properly. Therefore, in certain pipeline configurations, it may be *impossible* to maintain both PS_INVOCATIONS_COUNT and PS_DEPTH_COUNT accurately.

9 Color Calculator (Output Merger)

Note: **The Color Calculator logic resides in the Render Cache backing Data Port (DAP) shared function**. It is described in this chapter as the Color Calc functions are naturally an extension of the 3D pipeline past the WM stage. See the DataPort chapter for details on the messages used by the Pixel Shader to invoke Color Calculator functionality.

The *Color Calculator* function within the Data Port shared function completes the processing of rasterized pixels after the pixel color and depth have been computed by the Pixel Shader. This processing is initiated when the pixel shader thread sends a Render Target Write message (see *Shared Functions*) to the Render Cache. (Note that a single pixel shader thread may send multiple Render Target Write messages, with the result that multiple render targets get updated). The pixel variables pass through a pipeline of fixed (yet programmable) functions, and the results are conditionally written into the appropriate buffers.

Pipeline Stage	Description
Alpha Test	Compare pixel alpha with reference alpha and conditionally discard pixel
Stencil Test	Compare pixel stencil value with reference and forward result to Buffer Update stage
Depth Test	Compare pix.Z with corresponding Z value in the Depth Buffer and forward result to Buffer Update stage
Color Blending	Combine pixel color with corresponding color in color buffer according to programmable function
Gamma Correction	Adjust pixel's color according to gamma function for SRGB destination surfaces.
Color Quantization	Convert "full precision" pixel color values to fixed precision of the color buffer format
Logic Ops	Combine pixel color logically with existing color buffer color (mutually exclusive with Color Blending)
Buffer Update	Write final pixel values to color and depth buffers or discard pixel without update

The following logic describes the high-level operation of the Pixel Processing pipeline:

```
PixelProcessing() {
        AlphaTest()
        DepthBufferCoordinateOffsetDisable
        StencilTest()
        DepthTest()
        ColorBufferBlending()
        GammaCorrection()
        ColorQuantization()
        LogicalOps()
        BufferUpdate()
}
```

9.1.1 Alpha Test

The Alpha Test function can be used to discard pixels based on a comparison between the incoming pixel's alpha value and the **Alpha Test Reference** state variable in COLOR_CALC_STATE. This operation can be used to remove transparent or nearly-transparent pixels, though other uses for the alpha channel and alpha test are certainly possible.

This function is enabled by the **Alpha Test Enable** state variable in COLOR_CALC_STATE. If ENABLED, this function compares the incoming pixel's alpha value (*pixColor.Alpha*) and the reference alpha value specified by via the **Alpha Test Reference** state variable in COLOR_CALC_STATE. The comparison performed is specified by the **Alpha Test Function** state variable in COLOR_CALC_STATE.

The **Alpha Test Format** state variable is used to specify whether Alpha Test is performed using fixed-point (UNORM8) or FLOAT32 values. Accordingly, it determines whether the **Alpha Reference Value** is passed in a UNORM8 or FLOAT32 format. If UNORM8 is selected, the pixel's alpha value will be converted from floating-point to UNORM8 before the comparison.

Pixels that pass the Alpha Test proceed for further processing. Those that fail are discarded at this point in the pipeline.

If **Alpha Test Enable** is DISABLED, this pipeline stage has no effect.

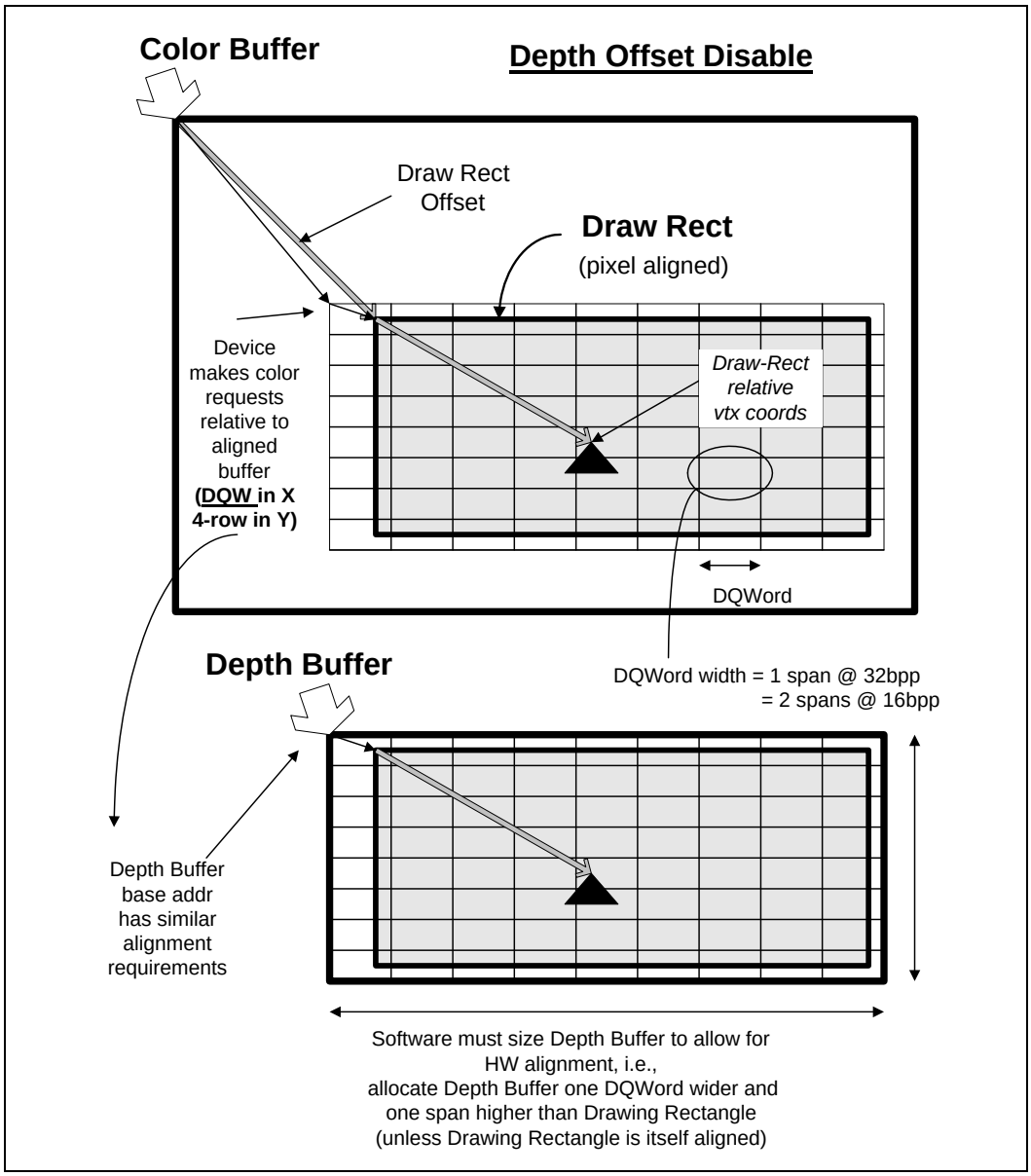
9.1.2 Depth Buffer Coordinate Offset Disable

There is a capability to effectively disable the application of the Drawing Rectangle coordinate offset for accesses to the Depth Buffer. This is controlled via the **Depth Buffer Coordinate Offset Disable** state variable in the 3DSTATE_DEPTH_BUFFER command. This capability exists in order to better support "front buffer rendering" where the Color Buffer is screen-sized (by definition) while the Depth Buffer does not have to be (i.e., it may be desired to have window-sized Depth Buffer to match a window-sized back buffer). Therefore the ability to offset only the Color (front) Buffer coordinate – and not the Depth Buffer coordinate – by the **Drawing Rectangle Origin X,Y** is desired. However, due to Color/Depth Buffer access alignment issues, the offset of the Depth Buffer X,Y coordinates can not be completely disabled – a few low-order bits of the **Drawing Rectangle Origin** must still be applied to provide some alignment of Color/Depth Buffer accesses.

The alignment restrictions require:

- 2 LSBs (when rendering 32-bit color) or 3 LSBs (when rendering 16-bit color) of the **Drawing Rectangle Origin X** are unconditionally applied to the Depth Buffer X coordinate. This corresponds to one 4x4 span or two 4x4 span alignment, respectively.
- 2 LSBs of **Drawing Rectangle Origin Y** are unconditionally applied to the Depth Buffer Y coordinate (i.e., 4-row co-alignment in Y)

Figure 9-1. Drawing Rectangle Offset



9.1.3 Stencil Test

The Stencil Test function can be used to discard pixels based on a comparison between the [**Backface**] **Stencil Test Reference** state variable and the pixel's stencil value. This is a general purpose function used for such effects as shadow volumes, per-pixel clipping, etc. The result of this comparison is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Stencil Test Enable** state variable. If ENABLED, the current stencil buffer value for this pixel is read.

Programming Notes:

- If the Depth Buffer is either undefined or does **not** have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT, **Stencil Test Enable** must be DISABLED.

A 2nd set of the stencil test state variables is provided so that pixels from back-facing objects, assuming they are not culled, can have a stencil test performed on them separate from the test for normal front-facing objects. The separate stencil test for back-facing objects can be enabled via the **Double Sided Stencil Enable** state variable. Otherwise, non-culled back-facing objects will use the same test function, mask and reference value as front-facing objects. The 2nd stencil state for back-facing objects is most commonly used to improve the performance of rendering shadow volumes which require a different stencil buffer operation depending on whether pixels rendered are from a front-facing or back-facing object. The backface stencil state removes the requirement to render the shadow volumes in 2 passes or sort the objects into front-facing and back-facing lists.

The remainder of this subsection describes the function in term of [**Backface**] <**state variable name>**. The Backface set of state variables are only used if Double Sided Stencil Enable is ENABLED and the object is considered back-facing. Otherwise the normal (front-facing) state variables are used.

This function then compares the [**Backface**] **Stencil Test Reference** value and the pixel's stencil value value after logically ANDing both values by [**Backface**] **Stencil Test Mask**. The comparison performed is specified by the [**Backface**] **Stencil Test Function** state variable. The result of the comparison is passed down the pipeline for use in the Stencil Buffer Update function. The Stencil Test function does not in itself discard pixels.

If **Stencil Test Enable** is DISABLED, a result of "stencil test passed" is propagated down the pipeline.

9.1.4 Depth Test

The Depth Test function can be used to discard pixels based on a comparison between the incoming pixel's depth value and the current depth buffer value associated with the pixel. This function is typically used to perform the "Z Buffer" hidden surface removal. The result of this pipeline function is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Depth Test Enable** state variable. If enabled, the pixel's ("source") depth value is first computed. After computation the pixel's depth

value is clamped to the range defined by **Minimum Depth** and **Maximum Depth** in the selected CC_VIEWPORT state. Then the current ("destination") depth buffer value for this pixel is read.

This function then compares the source and destination depth values. The comparison performed is specified by the **Depth Test Function** state variable.

The result of the comparison is propogated down the pipeline for use in the subsequent Depth Buffer Update function. The Depth Test function does not in itself discard pixels.

If **Depth Test Enable** is DISABLED, a result of "depth test passed" is propagated down the pipeline.

Programming Notes:

- Enabling the Depth Test function without defining a Depth Buffer is UNDEFINED.

9.1.5 Pre-Blend Color Clamping

Pre-Blend Color Clamping, controlled via **Pre-Blend Color Clamp Enable** and **Color Clamp Range** states in COLOR_CALC_STATE, is affected by the enabling of Color Buffer Blend as described below.

The following table summarizes the requirements involved with Pre-/Post-Blend Color Clamping.

Blending	RT Format	Pre-Blend Color Clamp	Post-Blend Color Clamp
Off	UNORM, UNORM_SRGB,YCRCB	Must be enabled with range = RT range or [0,1] (same function)	n/a, state ignored
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	n/a, state ignored
	FLOAT (except for R11G11B10_FLOAT)	Must be enabled (with any desired range)	n/a, state ignored
	R11G11B10_FLOAT	Must be enabled with either [0,1] or RT range	n/a, state ignored
	UINT, SINT	State ignored, implied clamp to RT range	n/a, state ignored
On (where permitted)	UNORM, UNORM_SRGB	Must be enabled with range = RT range or [0,1] (same function)	Must be enabled with range = RT range or [0,1] (same function)
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	Must be enabled with range = RT range or [-1,1] (same function)

Blending	RT Format	Pre-Blend Color Clamp	Post-Blend Color Clamp
	FLOAT (except for R11G11B10_FLOAT)	Can be disabled or enabled (with any desired range)	Must be enabled (with any desired range)
	R11G11B10_FLOAT	Can be disabled or enabled (with any desired range)	Must be enabled with either [0,1] or RT range

Note regarding Multiple RenderTargets (MRTs): There is only one set of Pre/Post-Blend Color Clamp state variables, and therefore they apply to all RTs (i.e., for each separate RT-Write DataPort message). If all RTs have the same format, then these controls can be programmed with the same flexibility as if there was only one RT. However, if the RTs can have differing formats, then software must ensure that the shared control settings make sense for each RT format. For example, specifying a pre-blend and post-blend clamp to RT-range will work for any combination of RT formats, while specifying a pre-blend clamp to [-1,1] when using a UNORM+SNORM MRT likely won't produce meaningful results in the UNORM RT.

9.1.5.1.1 Pre-Blend Color Clamping when Blending is Disabled

The clamping of source color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all source color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed.

Programming Notes:

- Given the possibility of writing UNPREDICTABLE values to the Color Buffer, it is expected and highly recommended that, when blending is disabled, software set **Pre-Blend Color Clamp Enable** to ENABLED and select an appropriate **Color Clamp Range**.
- When using SINT or UINT rendertarget surface formats, **Blending must** be DISABLED. The **Pre-Blend Color Clamp Enable** and **Color Clamp Range** fields are ignored, and an implied clamp to the rendertarget surface format is performed.

9.1.5.1.2 Pre-Blend Color Clamping when Blending is Enabled

The clamping of source, destination and constant color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all these color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed on these color components prior to blending.

9.1.6 Color Buffer Blending

The Color Buffer Blending function is used to combine one or two incoming "source" pixel color+alpha values with the "destination" color+alpha read from the corresponding location in a RenderTarget.

Blending is enabled on a global basis by the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE). If DISABLED, Blending and Post-Blend Clamp

functions are disabled for all RenderTargets, and the pixel values (possibly subject to Pre-Blend Clamp) are passed through unchanged.

If the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE) is ENABLED, then the RenderTarget's **Color Blend Enable** bit (in SURFACE_STATE) is used to determine if Blending is enabled or disabled. Note that each RenderTarget has its own "local" Color Blend Enable state, so in Multi-RenderTarget scenarios some RTs may have Blending enabled and other RTs may have Blending disabled.

DevBW-A,B Errata: The Color Blend Enable bit in SURFACE_STATE is not used, and acts as if it is ENABLED for each RenderTarget. Blending is enabled or disabled only a a global basis by **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE)

Programming Notes:

- Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
- Dual source blending:
  - **[DevBW, DevCL-A]** Not supported
  - **[DevCL-B]:** The DataPort only supports dual source blending with a SIMD8-style message.
- Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in *Sampling Engine*. Blending must be disabled on a RenderTarget if blending is not supported.

The incoming "source" pixel values are modulated by a selected "source" blend factor, and the possibly gamma-decorrected "destination" values are modulated by a "destination" blend factor. These terms are then combined with a "blend function". In general:

```
src_term = src_blend_factor * src_color
dst_term = dst_blend_factor * dst_color
color output = blend_function( src_term, dst_term)
```

If there is no alpha value contained in the Color Buffer, a default value of 1.0 is used and, correspondingly, there is no alpha component computed by this function.

[DevCL-B]: Dual Source Blending: When using "Dual Source" Render Target Write messages, the Source1 pixel color+alpha passed in the message can be selected as a src/dst blend factor. See Table 9-1. In single-source mode, those blend factor selections are invalid. If SRC1 is included in a src/dst blend factor and a DualSource RT Write message is not utilized, results are UNDEFINED. Also, it is UNDEFINED to utilize a DualSource RT Write message when Blending is disabled.

The blending of the color and alpha components is controlled with two separate (color and alpha) sets of state variables. However, if the **Independent Alpha Blend Enable** state variable in COLOR_CALC_STATE is DISABLED, then the "color" (rather than "alpha") set of state variables is used for both color and alpha. Note that this is the only use of the **Independent Alpha Blend Enable** state – it does not control whether Blending occurs, only how.

The following table describes the color source and destination blend factors controlled by the **Source [Alpha] Blend Factor** and **Destination [Alpha] Blend Factor** state variables in COLOR_CALC_STATE. Note that the blend factors applied to the R,G,B channels are always controlled by the **Source / Destination Blend Factor**, while the

blend factor applied to the alpha channel is controlled either by **Source / Destination Blend Factor** or **Source / Destination Alpha Blend Factor**.

Table 9-1. Color Buffer Blend Color Factors

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_DST_COLOR	rtN.r, rtN.g, rtN.b, rtN.a
BLENDFACTOR_INV_DST_COLOR	1.0-rtN.r, 1.0-rtN.g, 1.0-rtN.b, 1.0-rtN.a
BLENDFACTOR_DST_ALPHA	rtN.a, rtN.a, rtN.a, rtN.a
BLENDFACTOR_INV_DST_ALPHA	1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a
BLENDFACTOR_CONST_COLOR	CC.r, CC.g, CC.b, CC.a
BLENDFACTOR_INV_CONST_COLOR	1.0-CC.r, 1.0-CC.g, 1.0-CC.b, 1.0-CC.a
BLENDFACTOR_CONST_ALPHA	CC.a, CC.a, CC.a, CC.a
BLENDFACTOR_INV_CONST_ALPHA	1.0-CC.a, 1.0-CC.a, 1.0-CC.a, 1.0-CC.a
BLENDFACTOR_SRC_ALPHA_SATURATE	f,f,f,1.0 where f = min(1.0 – rtN.a, oN.a)

The following table lists the supported blending operations defined by the **Color Blend Function** state variable and the **Alpha Blend Function** state variable (when in independent alpha blend mode).

Table 9-2. Color Buffer Blend Functions

Blend Function	Operation (for each color component)
BLENDFUNCTION_ADD	SrcColor*SrcFactor + DstColor*DstFactor
BLENDFUNCTION_SUBTRACT	SrcColor*SrcFactor - DstColor*DstFactor
BLENDFUNCTION_REVERSE_SUBTRACT	DstColor*DstFactor - SrcColor*SrcFactor
BLENDFUNCTION_MIN	min (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "min" function.
BLENDFUNCTION_MAX	max (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "max" function.

9.1.6.1 3DSTATE_CONSTANT COLOR

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

The 3DSTATE_CONSTANT_COLOR command is used to specify the Constant Color used in Color Buffer Blending. It is a non-pipelined command.

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_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h 3DSTATE_CONSTANT_COLOR Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 3h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Blend Constant Color Red** Project: All Format: IEEE_Float FormatDesc This field specifies the Red channel of the Constant Color used in Color Buffer Blending.

3DSTATE_CONSTANT_COLOR		
2	31:0	**Blend Constant Color Green** Project: All Format: IEEE_Float FormatDesc This field specifies the Green channel of the Constant Color used in Color Buffer Blending.
3	31:0	**Blend Constant Color Blue** Project: All Format: IEEE_Float FormatDesc This field specifies the Blue channel of the Constant Color used in Color Buffer Blending.
4	31:0	**Blend Constant Color Alpha** Project: All Format: IEEE_Float FormatDesc This field specifies the Alpha channel of the Constant Color used in Color Buffer Blending.

9.1.7 Post-Blend Color Clamping

(See *Pre-Blend Color Clamping* above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

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

9.1.8 Color Quantization

[This is considered an implementation-specific topic, covered in the detailed hardware design documents.]

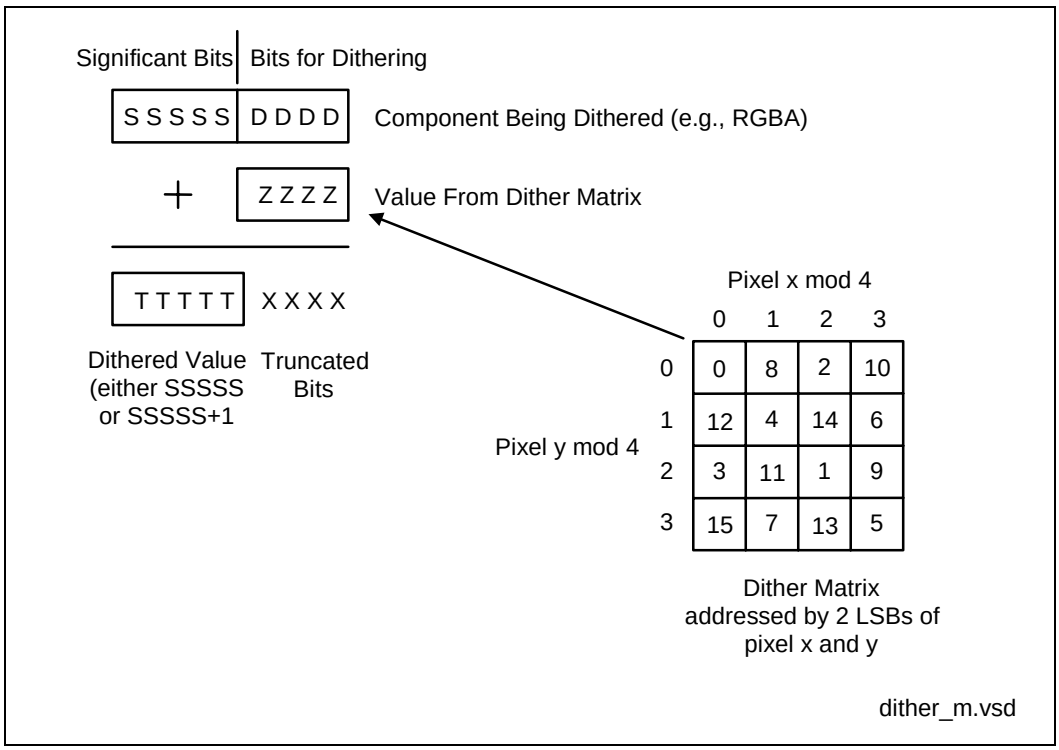
9.1.9 Dithering

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer's eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel's X and Y screen coordinate. The pixel's X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel's screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel's screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.

Figure 9-2. Dithering Process (5-Bit Example)



9.1.10 Buffer Update

The Buffer Update function is responsible for updating the pixel's Stencil, Depth and Color Buffer contents based upon the results of the Stencil and Depth Test functions. Note that Kill Pixel and/or Alpha Test functions may have already discarded the pixel by this point.

9.1.10.1 Stencil Buffer Updates

If and only if stencil testing is enabled, the Stencil Buffer is updated according to the **Stencil Fail Op**, **Stencil Pass Depth Fail Op**, and **Stencil Pass Depth Pass Op** state (or their backface counterparts if **Double Sided Stencil Enable** is ENABLED

and the pixel is from a back-facing object) and the results of the Stencil Test and Depth Test functions.

Stencil Fail Op and **Backface Stencil Fail Op** specify how/if the stencil buffer is modified if the stencil test fails. **Stencil Pass Depth Fail Op** and **Backface Stencil Pass Depth Fail Op** specify how/if the stencil buffer is modified if the stencil test passes but the depth test fails. **Stencil Pass Depth Pass Op** and **Backface Stencil Pass Depth Pass Op** specify how/if the stencil buffer is modified if both the stencil and depth tests pass. The operations (on the stencil buffer) that are to be performed under one of these (mutually exclusive) conditions is summarized in the following table.

Table 9-3. Stencil Buffer Operations

Stencil Operation	Description
STENCILOP_KEEP	Do not modify the stencil buffer
STENCILOP_ZERO	Store a 0
STENCILOP_REPLACE	Store the *StencilTestReference* reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible wrap around to 0)
STENCILOP_DECR	Decrement (possible wrap to max value)
STENCILOP_INVERT	Logically invert the stencil value

Any and all writes to the stencil portion of the depth buffer are enabled by the **Stencil Buffer Write Enable** state variable.

When writes are enabled, the **Stencil Buffer Write Mask** and **Backface Stencil Buffer Write Mask** state variables provide an 8-bit mask that selects which bits of the stencil write value are modified. Masked-off bits (i.e., mask bit == 0) are left unmodified in the Stencil Buffer.

Programming Notes:

- If the Depth Buffer does **not** have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT, **Stencil Buffer Write Enable** must be DISABLED.
- The Stencil Buffer can be written even if depth buffer writes are disabled via **Depth Buffer Write Enable**.

9.1.10.2 Depth Buffer Updates

Any and all writes to the Depth Buffer are enabled by the **Depth Buffer Write Enable** state variable. If there is no Depth Buffer, writes must be explicitly disabled with this state variable, or operation is UNDEFINED.

If depth testing is disabled or the depth test passed, the incoming pixel's depth value is written to the Depth Buffer. If depth testing is enabled and the depth test failed, the pixel is discarded – with no modification to the Depth or Color Buffers (though the Stencil Buffer may have been modified).

9.1.10.3 Color Gamma Correction

Computed RGB (not A) channels can be gamma-corrected prior to update of the Color Buffer.

This function is automatically invoked whenever the destination surface (render target) has an SRGB format (see surface formats in *Sampling Engine*). For these surfaces, the computed RGB values are converted from gamma=1.0 space to gamma=2.4 space by applying a ^(2.4) exponential function.

9.1.10.4 Color Buffer Updates

Finally, if the pixel has not been discarded by this point, the incoming pixel color is written into the Color Buffer. The **Surface Format** of the color buffer indicates which channel(s) are written (e.g., R8G8_UNORM are written with the Red and Green channels only). The **Color Buffer Component Write Disables** from the Color Buffer's SURFACE_STATE provide an independent write disable for each channel of the Color Buffer.

9.2 Pixel Pipeline State Summary

9.2.1 COLOR_CALC_STATE

The following COLOR_CALC_STATE definition applies to devices.

Dword	Bit	Description
0	31	**Stencil Test Enable:** Enables StencilTest function of the Pixel Processing pipeline. **[DevBW,DevCL-A] Errata:** See relevant errata in **Depth Test Enable** below. **Programming Notes:** • It is UNDEFINED to enable stencil test if a Stencil Buffer is not defined (i.e., when operating in 16bpp Depth mode) • If any of the render targets are YUV format, this field must be disabled. Format = Enable
	30:28	**Stencil Test Function:** This field specifies the comparison function used in the (front face) StencilTest function. Format = 3D_CompareFunction 0h: COMPAREFUNCTION_ALWAYS: 1h: COMPAREFUNCTION_NEVER: 2h: COMPAREFUNCTION_LESS: 3h: COMPAREFUNCTION_EQUAL: 4h: COMPAREFUNCTION_LEQUAL: 5h: COMPAREFUNCTION_GREATER: 6h: COMPAREFUNCTION_NOTEQUAL: 7h: COMPAREFUNCTION_GEQUAL:
	27:25	**Stencil Fail Op:** This field specifies the operation to perform on the Stencil Buffer when the (front face) stencil test fails. **Note:** if all three stencil ops (Stencil Fail, Stencil Pass Depth Fail, and Stencil Pass Depth Pass) are KEEP, ZERO, or REPLACE, the stencil buffer is not read. Format = 3D_StencilOperation = 0 = STENCILOP_KEEP 1 = STENCILOP_ZERO 2 = STENCILOP_REPLACE 3 = STENCILOP_INCRSAT 4 = STENCILOP_DECRSAT 5 = STENCILOP_INCR 6 = STENCILOP_DECR 7 = STENCILOP_INVERT
	24:22	**Stencil Pass Depth Fail Op :** This field specifies the operation to perform on the Stencil Buffer when the (front face) stencil test passes but the depth pass fails. Format = 3D_StencilOperation (see **Stencil Fail Op**)

Dword	Bit	Description
	21:19	**Stencil Pass Depth Pass Op :** This field specifies the operation to perform on the Stencil Buffer when the (front face) stencil test passes and the depth pass passes (or is disabled). Format = 3D_StencilOperation (see **Stencil Fail Op**)
	18	**Stencil Buffer Write Enable:** Enables writes to the Stencil Buffer. If **Stencil Test Enable** is disabled, writes to the stencil buffer are disabled independent of the setting of this field. **[DevBW,DevCL-A] Errata:** See relevant errata in **Depth Test Enable** below. **Programming Notes:** • When operating without a Stencil Buffer (i.e., when operating in 16bpp Depth mode), it is UNDEFINED to enable stencil writes via this field. Format = Enable
	17:16	Reserved : MBZ
	15	**Double Sided Stencil Enable:** Enable doubled sided stencil operations. **Programming Notes:** • Back-facing primitives have a vertex winding order opposite to the currently selected **Front Winding** state. • Culling of primitives is not affected by the double sided stencil state • Back-facing primitives will be rendered, honoring all current device state, as though it were a front-facing primitive with no implicitly overloaded state. Format = Boolean 0 = FALSE: Double Sided Stencil Disabled 1 = TRUE: Double Sided Stencil Enabled
	14:12	**BackFace Stencil Test Function:** This field specifies the comparison function used in the StencilTest function. Format = 3D_CompareFunction 0h: COMPAREFUNCTION_ALWAYS: 1h: COMPAREFUNCTION_NEVER: 2h: COMPAREFUNCTION_LESS: 3h: COMPAREFUNCTION_EQUAL: 4h: COMPAREFUNCTION_LEQUAL: 5h: COMPAREFUNCTION_GREATER: 6h: COMPAREFUNCTION_NOTEQUAL: 7h: COMPAREFUNCTION_GEQUAL:

Dword	Bit	Description
	11:9	**Backface Stencil Fail Op:** This field specifies the operation to perform on the Stencil Buffer when the stencil test fails. Format = 3D_StencilOperation = 0 = STENCILOP_KEEP 1 = STENCILOP_ZERO 2 = STENCILOP_REPLACE 3 = STENCILOP_INCRSAT 4 = STENCILOP_DECRSAT 5 = STENCILOP_INCR 6 = STENCILOP_DECR 7 = STENCILOP_INVERT
	8:6	**Backface Stencil Pass Depth Fail Op :** This field specifies the operation to perform on the Stencil Buffer when the stencil test passes but the depth pass fails. Format = 3D_StencilOperation (see **Stencil Fail Op**)
	5:3	**Backface Stencil Pass Depth Pass Op:** This field specifies the operation to perform on the Stencil Buffer when the stencil test passes and the depth pass passes (or is disabled). Format = 3D_StencilOperation (see **Stencil Fail Op**)
	2:0	Reserved : MBZ
1	31:24	**Stencil Reference Value:** This field specifies the stencil reference value to compare against in the (front face) StencilTest function. Format = U8.0
	23:16	**Stencil Test Mask:** This field specifies a bit mask applied to stencil test values. Both the stencil reference value and value read from the stencil buffer will be logically ANDed with this mask before the stencil comparison test is performed. Format = U8
	15:8	**Stencil Write Mask:** This field specifies a bit mask applied to stencil buffer writes. Only those stencil buffer bits corresponding to bits set in this mask will be modified. Format = U8
	7:0	**BackFace Stencil Reference Value:** This field specifies the stencil reference value to compare against in the StencilTest function. Format = U8.0
2	31:24	**Backface Stencil Test Mask:** This field specifies a bit mask applied to backface stencil test values. Both the stencil reference value and value read from the stencil buffer will be logically ANDed with this mask before the stencil comparison test is performed. Format = U8
	23:16	**Backface Stencil Write Mask:** This field specifies a bit mask applied to backface stencil buffer writes. Only those stencil buffer bits corresponding to bits set in this mask will be modified. Format = U8

Dword	Bit	Description
	15	**Depth Test Enable:** Enables the DepthTest function of the Pixel Processing pipeline. **[DevBW,DevCL-A] Errata:** Software must issue a PIPE_CONTROL command with the **Write Cache Flush Enable** set before transitioning from write-only depth/stencil mode (**Depth Test Enable** and **Stencil Test Enable** both DISABLED and **Depth Buffer Write Enable** or **Stencil Buffer Write Enable** ENABLED) to read/write depth/stencil mode (**Depth Test Enable** or **Stencil Test Enable** ENABLED), otherwise operation is UNDEFINED. **Programming Notes:** • If any of the render targets are YUV format, this field must be disabled. Format = Enable
	14:12	**Depth Test Function:** Specifies the comparison function used in DepthTest function. **Note:** if the Depth Test Function is ALWAYS or NEVER, the depth buffer is not read. Format = 3D_DepthTestFunction 0h: COMPAREFUNCTION_ALWAYS 1h: COMPAREFUNCTION_NEVER 2h: COMPAREFUNCTION_LESS 3h: COMPAREFUNCTION_EQUAL 4h: COMPAREFUNCTION_LEQUAL 5h: COMPAREFUNCTION_GREATER 6h: COMPAREFUNCTION_NOTEQUAL 7h: COMPAREFUNCTION_GEQUAL
	11	**Depth Buffer Write Enable:** Enables writes to the Depth Buffer. **[DevBW,DevCL-A] Errata:** See relevant errata in **Depth Test Enable** above. **Programming Notes:** • A Depth Buffer must be defined before enabling writes to it, or operation is UNDEFINED. Format = Enable
	10:1	Reserved : MBZ
	0	**Logic Op Enable:** Enables the LogicOp function of the Pixel Processing pipeline. **[DevBW,DevCL-A] Errata:** See relevant errata in Color Buffer Blend Enable description below. **Programming Notes:** • Enabling LogicOp and Color Buffer Blending at the same time is UNDEFINED Format = Enable
3	31:16	Reserved : MBZ
	15	**Alpha Test Format** This field selects the format for Alpha Reference Value and the format in which Alpha Test is performed. **Programming Notes:** • If the render target format is UNORM, this field must be set to ALPHATEST_UNORM8. 0 = ALPHATEST_UNORM8 1 = ALPHATEST_FLOAT32

Dword	Bit	Description
	14	Reserved : MBZ
	13	**Independent Alpha Blend Enable:** When enabled, the other fields in this instruction control the combination of the alpha components in the Color Buffer Blend stage. When disabled, the alpha components are combined in the same fashion as the color components. **Note:** See **Source Blend Factor** for a [**DevBW,DevCL] Erratum** which may require this field to be ENABLED Format = Enable
	12	**Color Buffer Blend Enable:** Enables the ColorBufferBlending (nee "alpha blending") function of the Pixel Processing Pipeline on a global basis. For Blending to be enabled, the **Color Blend Enable** bit of the RenderTarget's SURFACE_STATE must also be ENABLED. (See *Color Buffer Blending*). **[DevBW-A,B] Errata:** The **Color Blend Enable** bit in SURFACE_STATE is not used, and acts as if it is ENABLED for each RenderTarget. Blending is enabled or disabled only a a global basis by this **Color Buffer Blend Enable** state variable. **[DevBW,DevCL-A] Errata:** Software must issue a PIPE_CONTROL command with the **Write Cache Flush Enable** set before transitioning from write-only color mode (**Color Buffer Blend Enable** and **LogicOp Enable** both DISABLED) to read/write color mode (**Color Buffer Blend Enable** or **LogicOp Enable** ENABLED), otherwise operation is UNDEFINED. **Programming Notes:** • Enabling LogicOp and ColorBufferBlending at the same time is UNDEFINED Format = Enable
	11	**Alpha Test Enable:** Enables the AlphaTest function of the Pixel Processing pipeline. **Programming Notes:** • Alpha Test can only be enabled if all render targets have a surface format of a UNORM or FLOAT type. • Alpha Test is applied independently on each render target by comparing that render target's alpha value against the alpha reference value. If the alpha test fails, the corresponding pixel write will be supressed only for that render target. The depth/stencil update will occur if alpha test passes for *any* render target. Format = Enable
	10:8	**Alpha Test Function :** This field specifies the comparison function used in the AlphaTest function Format = 3D_CompareFunction 0h: COMPAREFUNCTION_ALWAYS: Always pass 1h: COMPAREFUNCTION_NEVER: Never pass 2h: COMPAREFUNCTION_LESS: Pass if the value is less than the reference 3h: COMPAREFUNCTION_EQUAL: Pass if the value is equal to the reference 4h: COMPAREFUNCTION_LEQUAL: Pass if the value is less than or equal to the reference 5h: COMPAREFUNCTION_GREATER: Pass if the value is greater than the reference 6h: COMPAREFUNCTION_NOTEQUAL: Pass if the value is not equal to the reference 7h: COMPAREFUNCTION_GEQUAL: Pass if the value is greater than or equal to the reference

Dword	Bit	Description
	7:0	Reserved : MBZ
4	31:5	**Color Calculator Viewport State Pointer:** Specifies the 32-byte aligned address offset of CC_VIEWPORT. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** CC_VIEWPORT data pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5]
	4:0	Reserved : MBZ
5	31	**Color Dither Enable:** Enables dithering of colors (including any alpha component) before they are written to the Color Buffer. Format = Enable
	30	**Round Disable Function Disable:** Disables the round-disable function of the color calculator. If this bit is zero, dithering is cancelled based on the data used by blend to avoid drift. If this bit is one, this is not done. Format = Disable **[DevBW]:** this bit must be set to zero.
	29:20	Reserved : MBZ
	19:16	**Logic Op Function:** This field specifies the function to be performed (when enabled) in the Logic Op stage of the Pixel Processing pipeline. Note that the encoding of this field is one less than the corresponding "R2_" ROP code defined in WINGDI.H, and is a rather contorted mapping of the OpenGL LogicOp encodings. However, this field was defined such that, when the 4 bits are replicated to 8 bits, they coincide with the ROP codes used in the Blter. **Note:** if the Logic Op Function does not depend on "D", the dest buffer is not read. Format = 3D_LogicOpFunction: 0h: LOGICOP_CLEAR BLACK; all 0's 1h: LOGICOP_NOR NOTMERGEPEN; NOT (S OR D) 2h: LOGICOP_AND_INVERTED MASKNOTPEN; (NOT S) AND D 3h: LOGICOP_COPY_INVERTED NOTCOPYPEN; NOT S 4h: LOGICOP_AND_REVERSE MASKPENNOT; S AND NOT D 5h: LOGICOP_INVERT NOT; NOT D 6h: LOGICOP_XOR XORPEN; S XOR D 7h: LOGICOP_NAND NOTMASKPEN; NOT (S AND D) 8h: LOGICOP_AND MASKPEN; S AND D 9h: LOGICOP_EQUIV NOTXORPEN; NOT (S XOR D) Ah: LOGICOP_NOOP NOP; D Bh: LOGICOP_OR_INVERTED MERGENOTPEN; (NOT S) OR D Ch: LOGICOP_COPY COPYPEN; S Dh: LOGICOP_OR_REVERSE MERGEPENNOT; S OR NOT D Eh: LOGICOP_OR MERGEPEN; S OR D Fh: LOGICOP_SET WHITE; all 1's

Dword	Bit	Description
	15	**Statistics Enable:** If ENABLED, the pixel pipeline will engage in statistics gathering. If DISABLED, statistics information associated with this FF stage will be left unchanged. **Programming Notes:** • If this field is enabled, **Statistics Enable** in WM_STATE should also be set, and when this field is disabled, **Statistics Enable** in WM_STATE should also be clear. Both functions contribute to the PS_DEPTH_COUNT, so having either one set without the other set will result in an UNPREDICTABLE value for PS_DEPTH_COUNT. Format = Enabled
	14:12	**Alpha Blend Function:** This field specifies the function used to combine the alpha components in the Color Buffer blend stage of the Pixel Pipeline when the *IndependentAlphaBlend* state is enabled. Format = 3D_ColorBufferBlendFunction : 0 = BLENDFUNCTION_ADD 1 = BLENDFUNCTION_SUBTRACT 2 = BLENDFUNCTION_REVERSE_SUBTRACT 3 = BLENDFUNCTION_MIN 4 = BLENDFUNCTION_MAX 5-7 = Reserved

Dword	Bit	Description
	11:7	**Source Alpha Blend Factor:** Controls the “source factor” in alpha Color Buffer Blending stage. **Note:** For the source/destination alpha blend factors, the encodings indicating “COLOR” are the same as the encodings indicating “ALPHA”, as the alpha component of the color is selected. See **Source Blend Factor** for [**DevBW,DevCL] Errata** Format = 3D_ColorBufferBlendFactor 00h: Reserved 01h: BLENDFACTOR_ONE 02h: BLENDFACTOR_SRC_COLOR 03h: BLENDFACTOR_SRC_ALPHA 04h: BLENDFACTOR_DST_ALPHA 05h: BLENDFACTOR_DST_COLOR 06h: BLENDFACTOR_SRC_ALPHA_SATURATE (See **Source Blend Factor** for [**DevBW,DevCL] Errata)** 07h: BLENDFACTOR_CONST_COLOR 08h: BLENDFACTOR_CONST_ALPHA 09h: Reserved 0Ah: Reserved 11h: BLENDFACTOR_ZERO 12h: BLENDFACTOR_INV_SRC_COLOR 13h: BLENDFACTOR_INV_SRC_ALPHA 14h: BLENDFACTOR_INV_DST_ALPHA 15h: BLENDFACTOR_INV_DST_COLOR 17h: BLENDFACTOR_INV_CONST_COLOR 18h: BLENDFACTOR_INV_CONST_ALPHA 19h: Reserved 1Ah: Reserved
	6:2	**Destination Alpha Blend Factor:** Controls the “destination factor” in alpha Color Buffer Blending stage. Format = 3D_ColorBufferBlendFactor See **Source Blend Factor** for [**DevBW,DevCL] Errata** Refer to **Source Alpha Blend Factor** for encodings.
	1:0	Reserved : MBZ

Dword	Bit	Description
6	31:29	**Color Blend Function:** This field specifies the function used to combine the color components in the ColorBufferBlending function of the Pixel Processing Pipeline. If **Independent Alpha Blend Enable** is disabled, this field will also control the blending of the alpha components in the ColorBufferBlending function. Format = 3D_ColorBufferBlendFunction 0 = BLENDFUNCTION_ADD 1 = BLENDFUNCTION_SUBTRACT 2 = BLENDFUNCTION_REVERSE_SUBTRACT 3 = BLENDFUNCTION_MIN 4 = BLENDFUNCTION_MAX
	28:24	**Source Blend Factor:** Controls the “source factor” in the ColorBufferBlending function. Format = 3D_ColorBufferBlendFactor **[DevBW,DevCL] Erratum**: Use of BLENDFACTOR_SRC_ALPHA_SATURATE for a source or dest blend factor yields an incorrect Alpha blend factor for R32G32B32A32_FLOAT RTs. The suggested SW workaround is (for at least R32G32B32A32_FLOAT RTs) to set **Independent Alpha Blend Enable** to ENABLED and program the Alpha Blend factor(s) to BLENDFACTOR_ONE as required – offering the same functionality. **[DevBW,DevCL] Erratum**: If (a) BLENDFACTOR_SRC_ALPHA_SATURATE is specified, and (b) either src or dest is contains a NaN value, then the R/G/B channels of the blend output are UNDEFINED. Refer to **Source Alpha Blend Factor** for encodings.
	23:19	**Destination Blend Factor:** Controls the “destination factor” in the ColorBufferBlending function. Format = 3D_ColorBufferBlendFactor See **Source Blend Factor** for [**DevBW,DevCL] Errata** Refer to **Source Alpha Blend Factor** for encodings.
	18:17	**X Dither Offset:** Specifies offset to apply to pixel X coordinate LSBs when accessing dither table. Format = U2
	16:15	**Y Dither Offset:** Specifies offset to apply to pixel Y coordinate LSBs when accessing dither table. Format = U2
	14:4	Reserved : MBZ

Dword	Bit	Description
	3:2	**Color Clamp Range:** Specifies the clamped range used in Pre-Blend and Post-Blend Color Clamp functions if one or both of those functions are enabled. Note that this range selection is shared between those functions. This field is ignored if both of the **Color Clamp Enables** are disabled. 0 = COLORCLAMP_UNORM: Clamp Range [0,1] 1 = COLORCLAMP_SNORM: Clamp Range [-1,1] 2 = COLORCLAMP_RTFORMAT: Clamp to the range of the RT surface format (Note: The Alpha component is clamped to FLOAT16 for R11G11B10_FLOAT format). 3 = Reserved **[DevBW,DevCL] Erratum**: If (a) **Post-Blend Color Clamp Enable** is set, and (b) COLORCLAMP_UNORM ([0,1]) or COLORCLAMP_SNORM ([-1,1]) range is specified, and (c) the rendertarget has a less-than-32-bit floating point format (float16 or R11G11B10_FLOAT), and (d) one of the color channels contains a NaN value, then the output is UNDEFINED (at very least the NaN is not preserved in the output).
	1	**Pre-Blend Color Clamp Enable:** This field specifies whether the source, destination and constant color channels are clamped prior to blending, regardless of whether blending is enabled. If DISABLED, no clamping is performed prior to blending. If ENABLED, all inputs to the blend function are clamped prior to the blend to the range specified by **Color Clamp Range**. **Programming Notes:** • See table in *Pre-Blending Color Clamp* subsection for programming restrictions as a function of RT format. • This field is ignored (treated as DISABLED) for UINT and SINT RT surface formats. Blending is not supported for those RT surface formats. The device will automatically clamp source color channels to the respective RT surface range. Format = Enable
	0	**Post-Blend Color Clamp Enable:** If blending is enabled, this field specifies whether the blending output channels are first clamped to the range specified by **Color Clamp Range**. Regardless of whether this clamping is enabled, the blending output channels will be clamped to the RT surface format just prior to being written. **Programming Notes:** • See table in *Pre-Blending Color Clamp* subsection for programming restrictions as a function of RT format. • This field is ignored (treated as DISABLED) for UINT and SINT RT surface formats. Blending is not supported for those RT surface formats. The device will automatically clamp source color channels to the respective RT surface range. • See **Color Clamp Range** for a [**DevBW,DevCL] Erratum** Format = Enable
7	31:0	**Alpha Reference Value:** This field specifies the alpha reference value to compare against in the Alpha Test function. If **Alpha Test Format** == ALPHATEST_UNORM8 Format = UNORM8 (upper 24 bits MBZ) If **Alpha Test Format** == ALPHATEST_FLOAT32 Format = IEEE_FP

9.2.2 CC_VIEWPORT

The viewport 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 2 DWords apart. The first element of the viewport state array is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31:0	**Minimum Depth:** Indicates the minimum depth. The interpolated or computed depth is clamped to this value prior to the depth test. Format = IEEE_Float
1	31:0	**Maximum Depth:** Indicates the maximum depth. The interpolated or computed depth is clamped to this value prior to the depth test. Format = IEEE_Float

9.3 Other Pixel Pipeline Functions

9.3.1 Statistics Gathering

If **Statistics Enable** is set in WM_STATE *and* in CC_STATE, the PS_DEPTH_COUNT register (see Memory Interface Registers in Volume Ia, *GPU*) will be incremented once for each pixel that passes the depth, stencil and alpha tests. Note that each of these tests is treated as passing if disabled. This count is accurate regardless of whether **Early Depth Test Enable** is set. In order to obtain the value from this register at a deterministic place in the primitive stream without flushing the pipeline, however, the 3DCONTROL command must be used. See the *3D Pipeline* chapter in this volume for details on 3DCONTROL.

[DevBW-A] Errata BWT008: PS_DEPTH_COUNT cannot be accurately read using PIPE_CONTROL. Attempting to do so will result in an UNDEFINED value being written out to the PIPE_CONTROL target address.

§§

10 Media and General Purpose Pipeline

10.1 Introduction

This section covers the programming details for the media (general purpose) fixed function pipeline. The **media pipeline** is positioned in parallel with the 3D fixed function pipeline. It is so named as its initial (and primary) usage is to provide media functionalities and it does have media specific fixed function capability. However, the fixed functions are designed to have the general capability of controlling the shared functions and resources, feeding generic threads to the Execution Units to be executed, and interacting with such generic threads during run time. The media pipeline can be used for non-media applications, and therefore, can also be referred to as the **general purpose pipeline**. *For the rest of this chapter, we will refer this fixed function pipeline as the media pipeline, keeping in mind its general purpose capability.*

Concurrency of the media pipeline and the 3D pipeline is not supported. In other words, only one pipeline can be activated at a given time. Switching between the two pipelines within a single context is supported using the MI_PIPELINE_SELECT command.

The followings are some media application examples that can be mapped onto the media pipeline. All these applications are functional; however, what level of performance can be achieved depends on the hardware configuration and is beyond the scope of this document.

- MPEG-2 decode acceleration with HWMC
- MPEG-2 decode acceleration with IS/IDCT and forward
- MPEG-2 decode acceleration with VLD and forward
- AVC decode acceleration with HWMC and forward including Loop Filter
- VC1 decode acceleration with HWMC and forward including Loop Filter
- Advanced deinterlace filter (motion detected or motion compensated deinterlace filter)
- Video encode acceleration (with various level of hardware assistant)

10.1.1 Terminologies

Term	Definition
AVC	Advanced Video Coding. An international video coding standard jointly developed by MPEG and ITU. It is also known as H.264 (ITU), or MPEG-4 Part 10 (MPEG).
Child Thread	A thread corresponding to a leaf-node or a branch-node in a thread generation hierarchy. All thread originated from kernels running on the GEN4 execution units are child threads.
EOB	End of Block. It is a 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer.
IDCT	Inverse Discrete Cosine Transform. It is the stage in the video decoding pipe between IQ and MC.
ILDB	In-loop Deblocking Filter – the deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe
IQ	Inverse Quantization. It is a stage in the video decoding pipe between IS and IDCT.
IS	Inverse Scan. It is a stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for MPEG-2.
IT	Inverse Integer Transform. It is the stage in AVC or VC1 video decoding pipe between IQ and MC.
MPEG	Motion Picture Expert Group. MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
MC	Motion Conpensation. It is part of the video decoding pipe.
MVFS	Motion Vector Field Selection – a four-bit field selecting reference fields for the motion vectors of the current macroblock.
PRT	Persistent Root Thread is in the context of Advanced Scheduler, where the thread supports midstream interruptability for fine grain context switch. A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread. Only one PRT is allowed in the system. Upon context switch interrupt, instead of proceeding to completion, a PRT can save its software context and terminate. Hardware is responsible of re-dispatching the incomplete PRT at context restore, and a PRT can continue operations from that previously left-over state.
Parent Thread	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Root Thread	A thread corresponding to a root-node in a thread generation hierarchy. In the GEN4 general-purpose pipeline, all threads originated from VFE unit are root threads.
Synchronized Root Thread	A root thread that is dispatched by TS upon a ‘dispatch root thread’ message.

Term	Definition
TS	Thread Spawner. It is the second (and the last) fixed function in the GEN4 general-purpose pipeline.
Unsynchronized Root Thread	A root thread that is automatically dispatched by TS.
VFE	Video Front End. It is the first fixed function in the GEN4 general-purpose pipeline.
VLD	Variable Length Decode. It is the first stage of the video decoding pipe that consists mainly of bit-wide operations. GEN4 supports hardware MPEG-2 VLD acceleration in the VFE fixed function stage.

10.2 Media Pipeline Overview

The media (general purpose) pipeline consists of two fixed function units: Video Front End (VFE) unit and Thread Spawner (TS) unit. VFE unit interfaces with the Command Streamer (CS), writes thread payload data into the Unified Return Buffer (URB) and prepares threads to be dispatched through TS unit. VFE unit also contains a hardware Variable Length Decode (VLD) engine for MPEG-2 video decode. TS unit is the only unit of the media pipeline that interfaces to the Thread Dispatcher (TD) unit for new thread generation. It is responsible of spawning root threads (short for the root-node parent threads) originated from VFE unit and spawning child threads (can be either a leaf-node child thread or a branch-node parent thread) originated from the Execution Units (EU) by a parent thread (can be a root-node or a branch-node parent thread).

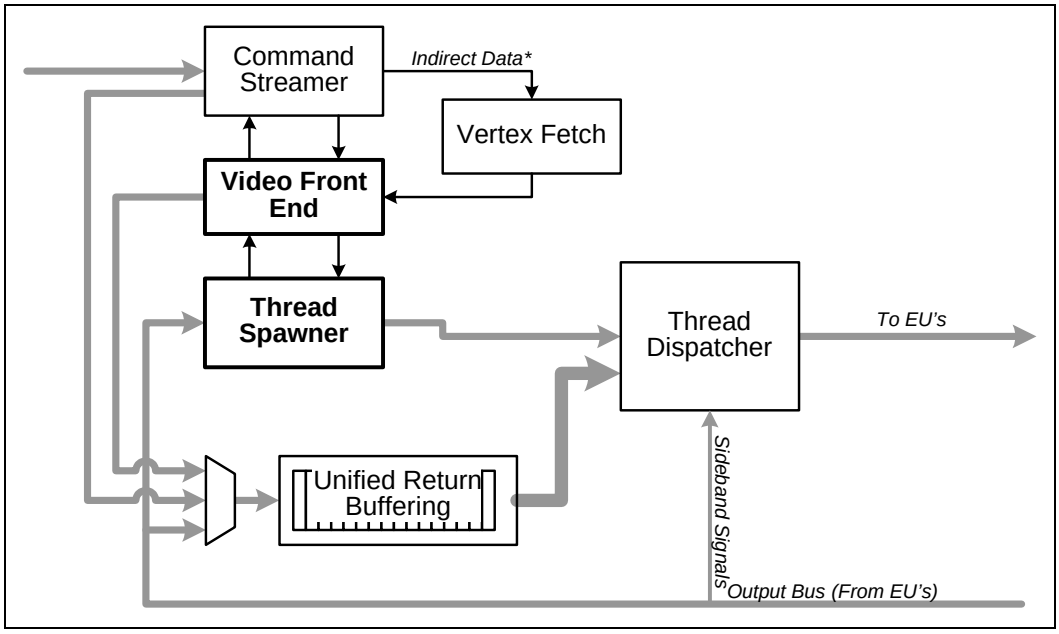
The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- VFE manages URB and only has write access to URB; TS does not interface to URB.
- When URB Constant Buffer is enabled, VFE forwards TS the URB Handler for the URB Constant Buffer received from CS.
- TS interfaces to TD; VFE does not.
- TS can have a message directed to it like other shared functions (and thus TS has a shared function ID), and it does not snoop the Output Bus as some other fixed functions in the 3D pipeline do.
- A root thread generated by the media pipeline can only have up to one URB return handle.
- If a root thread has a URB return handle, VFE creates the URB handle for the payload to initiating the root thread and also passes it alone to the root thread as the return handle. The root thread then uses the same URB handle for child thread generation.
- If URB Constant Buffer is enabled and an interface descriptor indicates that it is also used for the kernel, TS requests TD to load constant data directly to the thread's register space. For root thread, constant data are loaded after R0 and before the data from the other URB handle. For child thread, as the R0 header is provided by the parent thread, Thread Spawner splits the URB handles from the parent thread into two and inserts the constant data after the R0 header.
- A root thread must terminate with a message to TS. A child thread should also terminate with a message to TS.
- High streaming performance of indirect media object load is achieved by utilizing the large vertex cache available in the Vertex Fetch unit (of the 3D pipeline).

[**DevBW**] **Erratum**: DevBW doesn't not have MPEG-2 VLD hardware. Therefore, software cannot use the VLD mode of the Media_Object command.

[**DevBW-A**] **Erratum**: Using vertex cache in Vertex Fetch unit to speed up streaming of indirect media data load is not available on DevBW-A. On DevBW-A, indirect media data are loaded directly from CS to VFE.

Figure 10-1. Top level block diagram of the Media Pipeline



10.3 Programming Media Pipeline

10.3.1 Command Sequence

Media pipeline uses a simple programming model. Unlike the 3D pipeline, it does not support pipelined state changes. Any state change requires an MI_FLUSH or PIPE_CONTROL command. When programming the media pipeline, it should be cautious to not use the pipelining capability of the commands described in the Graphics Processing Engine chapter.

The basic steps in programming the media pipeline are listed below. Some of the steps are optional; however, the order must be followed strictly. Some usage restrictions are highlighted for illustration purpose. For details, reader should refer to the respective chapters for these commands.

- Special Requirements for Each Context Initialization
    - Always initialize the URB fence (with a URB_FENCE command) before the first pipeline select command (PIPELINE_SELECT).
    - Always initialize the pipeline state pointer (with a STATE_BASE_ADDRESS command) before the first pipeline select command.

- Step 1: MI_FLUSH/PIPE_CONTROL
  - This step is mandatory.
  - Programmer may choose not to flush certain caches to improve performance.
  - Multiple such commands in step 1 are allowed, but not recommended for performance reason.
  - [**DevBW-B, DevBW-C, DevCLN**]
    - MI_LOAD_REGISTER_IMM
    - It is used to load an MMIO register to disable the vertex cache for indirect media object load. The register is 0x2124 and the bit is 15.
      - Address = 0x2124
      - Data = 0x10000000
    - MI_FLUSH
    - MI_LOAD_REGISTER_IMM
    - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
    - IF present, it is used to load an MMIO register to enable the vertex cache for indirect media object load. The register is 0x2124 and the bit is 15.
      - Address = 0x2124
      - Data = 0x10001000

- Step 1.5: SF_STATE + URB_FENCE **Errata**
  - When switching from 3D context to Media context, the following sequence must be sent before the PIPE_SELECT command.
    - SF_STATE command must be sent down to set the “Number of URB Entries” to “0”.
    - URB_FENCE command must be sent down to set the “URB Fence” for all 3D units to “0”, including CS, VS, GS, CL, ans SF.

- Step 2: PIPELINE_ SELECT
  - This step is optional. This command can be omitted if it is known that within the same context media pipeline was selected before Step 1.
  - Multiple such commands in step 2 are allowed, but not recommended for performance reason.
  - If this command is issued, it must be followed by a URB_FENCE command (step 3).
- Step 3: URB_FENCE
  - This step is optional. This command can be omitted if URB fence needs not to be changed. However, as mentioned above, if a PIPELINE_SELECT command is issued, this command is then required.
  - If present, only one URB_FENCE command in step 3 is allowed. Hardware behavior is undefined if more than one URB_FENCE commands are issued in this step.

- Step 4: Configuring pipeline states
  - STATE_BASE_ADDRESS
    - This command is mandatory for this step (i.e. at least one).
    - Multiple such commands in this step are allowed. The last one overwrites previous ones.
    - This command must precede any other state commands below.
    - Particularly, the fields **Indirect Object Base Address** and **Indirect Object Access Upper Bound** are used to control indirect object load.
    - *Note: This command may be inserted before (and after) any commands listed in the previous steps (Step 1 to 3). For example, this command may be placed in the ring buffer while the others are put in a batch buffer.*
  - The following state commands can be issued in arbitrary order.
  - MEDIA_STATE_POINTERS
    - This command is mandatory for this step (i.e. at least one).
    - Multiple such commands in this step are allowed. The last one overwrites previous ones.
  - CS_URB_STATE
    - This command is optional for this step. Note that if CS_URB_STATE command is present, there will be at least one MEDIA_STATE_POINTERS command in this step (as mentioned above).
    - Multiple such commands in this step are allowed. The last one overwrites previous ones.
    - If present, “Number of URB Entries” must be 0 if no URB entry is allowed to CS by URB_FENCE command.
    - “Number of URB Entries” must be set to 1 as media pipeline does not support pipelined CONSTANT_BUFFER command (see step 5).
  - STATE_PREFETCH
    - This command is optional for this step.
  - STATE_SIP
    - This command is optional for this step. It is only required when SIP is used by the kernels.
  - 3DSTATE_VERTEX_ELEMENTS ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
    - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.

- If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
  - Two elements need to be programmed
  - Vertex Element 0
    - Vertex Buffer Index = 0
    - Valid = True
    - Surface Format = 0x002
    - Source Element Offset = 0x0
    - Component Control 0,1,2,3 = 0x1
    - Destination Offset = 0x0
  - Vertex Element 1
    - Vertex Buffer Index = 0
    - Valid = True
    - Surface Format = 0x002
    - Source Element Offset = 0x10
    - Component Control 0,1,2,3 = 0x1
    - Destination Offset = 0x10

- 3DSTATE_VERTEX_BUFFERS ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
  - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
  - If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
    - Only 1 vertex buffer
    - Buffer Access Type : Vertex Data
    - Buffer Pitch : 0x20
    - Buffer Start Address : <Indirect Data Address>
      - When VFE is in Generic Mode, the vertex buffer base address can be byte aligned. The restriction is that indirect data size of each MEDIA_OBJECT command must be a multiple of 32 bytes.
      - When VFE is either in VLD mode or IS mode, the indirect data size may not be multiple of 32 bytes (that's OK). However, it is required that the vertex buffer to be programmed to be 32-byte aligned. All indirect data must be included in the vertex buffer programmed.
    - Max index is always set to 0 (i.e., disabled)

- Step 5: CONSTANT_BUFFER
  - This step is optional. However, it is required (as a software workaround) when 3DPRIMITIVE commands are used subsequently to load indirect object data.

    - If present, only one such command is allowed. Hardware behavior is undefined if more than one CONSTANT_BUFFER commands are issued in the program sequence without a FLUSH in between.
- Step 6: Primitive commands
    - 3DPRIMITIVE ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
        - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
        - If present, this command must precede one or many MEDIA_OBJECT commands. If more than one MEDIA_OBJECT commands are followed, the indirect object data for these commands must be stored in memory contiguously (with certain 32-byte aligned overlaps allowed, see XXX for details).
        - If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
            - Sequential access for the Vertex Buffer
            - Primitive topology type is 0x01h = PointList
            - Vertex Count = Size of the block to transfer for the media indirect command in 32 byte quantities.
            - Start Vertex Location = 0
            - Base Vertex Location = 0
    - MEDIA_OBJECT
        - This step is optional, but it doesn't make practical sense not issuing media primitive commands after being through previous steps to set up the media pipeline.
        - Multiple such commands in step 6 can be issued to continue processing media primitives.

Programming Notes on Improving Indirect Media Object Load Performance [DevBW-C, DevCL]: The large vertex cache is used to stream indirect media object loads for one or many MEDIA_OBJECT commands. By grouping multiple such commands together significant streaming performance can be achieve. Here is an example.

- 1 Vertex Buffer programmed with 2 Vertex components
- Vertex format is fixed to A32R32B32G32_UINT, this format is left untouched by the vertex fetch

Here the number of vertices equal to the total size to be transferred for MEDIA_OBJECT commands in 32-byte chunks If the first MEDIA_OBJECT command transfers indirect data size of 4 64 byte quantities, the number of vertices would be 8, If the second MEDIA_OBJECT command to be transferred and the total size is 8 64 byte quantities, number of vertices is 16.

However, using the Vertex Buffer in sequential mode as described above does post a restriction that data for multiple MEDIA_OBJECT commands sharing the same 3DPRIMITIVE command must be stored in memory sequentially. When data are not stored sequentially in memory, there are several approaches as listed below. Certain experiments may be required in order to find which approach provides the best performance for a given application.

- Preceding each MEDIA_OBJECT command with one 3DPRIMITIVE command.

- Grouping several 3DPRIMITIVE commands together followed by the MEDIA_OBJECT commands using the fetched data from the 3DPRIMITIVE commands.
- Using indexed vertex buffer to gather indirect media object data from non-contiguous memory locations.

As a side effect, when vertex cache is used for media indirect object load, the statistical counters in the VF unit may be affected during media operations. When 3D operations and media operations are from different contexts, this side effect is not an issue as the statistical counters are context save/restored. However, if 3D and media operations are mixed within one context, it is advisable to turn off the statistical counters before entering media operation (using vertex cache for indirect object load) and turn them back on before returning to 3D operations. This can be achieved using the 3DSTATE_VF_STATISTICS command.

10.3.2 Interrupt Latency

Command Streamer is capable of context switching between primitive commands.

For all independent threads, it is not much a problem. The interrupt latency is dictated by the longest command that is likely to have the largest number of threads. For VLD mode, such a command may be corresponding to a largest slice in a high definition video frame. This is application dependent, there are not much host software can do. For Generic mode, programmer should consider to constrain the compute workload size of each thread.

In modes with child threads, a root thread may be persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. Software designer should partition the workload to restrict the duration of each root thread. For example, this may be achieved by partitioning a video frame and assigning separate primitive commands for different data partitions.

In modes with synchronized root threads, a synchronized root thread is dependent on a previous root or child thread. This means context switch is not allowed between the primitive command for the synchronized root thread and the one for the depending thread. So no command queue arbitration should be allowed between them. Software designer should also restrict the duration of such non-interruptible primitive command segments.

10.4 Video Front End Unit

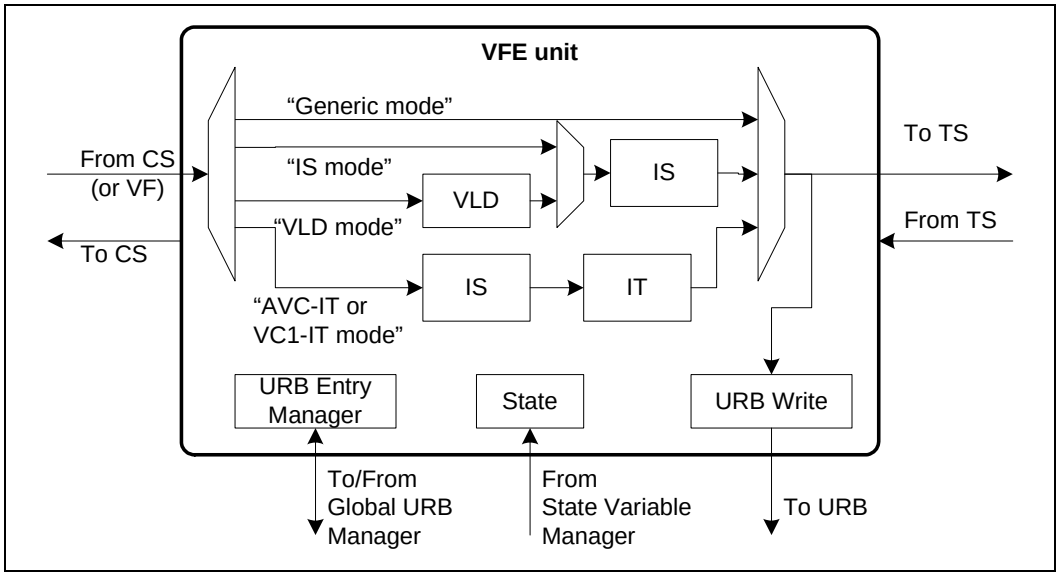
The Video Front End unit is the first fixed function unit in the media pipeline. It processes MEDIA_OBJECT commands to generate root threads by preparing the control (including interface descriptor pointers) and payload (data pushed into the GRF) for the root threads.

VFE supports three modes of operation: Generic mode, Inverse Scan mode and VLD mode.

- **Generic mode**: In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. There is no application specific hardware enabled in this mode.
- **IS (Inverse Scan) mode**: The IS mode is a special mode for video decoding when off-host IDCT acceleration is supported by kernels running on GEN4 execution units.
- **VLD mode**: It is a special mode for video decoding when MPEG-2 off-host VLD acceleration is supported by GEN4 hardware.

The following figure illustrates the three modes of operation. The details can be found in the rest of the sections.

Figure 10-2. VFE Functional Blocks and Modes of Operations



MEDIA_STATE_POINTERS command configures VFE in one of the three modes using. Mode switching requires media pipeline state change.

10.4.1 Interfaces

VFE unit acquires its states from Sate Variable Manager, accesses URB handles from the Global URB Manager, receives state and primitive commands from CS unit, writes thread payloads to URB, and sends new thread to TS unit. It does not directly interface to Thread Dispatcher. When VFE is ready for a thread, it sends the interface descriptor pointer for the thread to TS.

10.4.1.1 Interface to Command Streamer

VFE interfaces to CS to acquire the control data, inline data and indirect data of MEDIA_OBJECT commands. The interface supports the throughput of a given mode of operation of VFE. For example, in VLD mode and IS mode, VFE consumes one dword at a time, one dword to the variable length decoder or one dword to the inverse-scan operator. In Generic mode, VFE is capable of a much higher throughput to push indirect data (as thread payload data) into URB. As throughput for indirect data is much higher than that of inline data, when large amount of user data need to be passed through VFE unit, if applicable, it is encouraged to use indirect object load.

10.4.1.2 Interface to Thread Spawner

When a new root thread is fully assembled by VFE, VFE passes to TS the interface descriptor pointer, the URB handle information, the debug information, etc. In response to this, TS processes the thread information and sends a thread request to TD.

VFE also transmits scratch memory base address received from State Variable Manager to TS, and passes on the Constant URB handle received from CS.

VFE receives URB handle dereference signal from TS.

10.4.1.3 Interface to State Variable Manager

State Variable Manager is responsible of fetching media state structure from memory. VFE only acquires its state variable upon the first primitive command. Therefore, host software is allowed to change media states before issuing primitive commands. As media pipeline does not support pipelined state change, a pipeline flush is required before any state change to make sure that there are no outstanding primitive commands in the pipeline.

10.4.1.4 Interface to Global URB Manager

VFE is responsible for managing URB handles for all root threads. Upon state change, VFE allocates URB handles through the Global URB Manager. VFE manages the URB handles in a circular buffer. URB handle referencing is in a strict order (taking from the head of the circular buffer), even though the handle dereferencing may occur out of order.

When starting a root thread, VFE reference one and only one URB handle, forwarding it to TS. TS then forwards this handle to TD for thread dispatching.

The URB handle for a root thread is used in two ways: (1) serving as buffer space for VFE to assemble thread payload, and (2) serving as the return URB buffer for the root thread to assemble child threads and their payload.

TS sends an indication to VFE when it is safe to dereference the URB handle, and VFE dereferences it. After a URB handle has been dereferenced, VFE can assign it to a new thread.

10.4.1.5 Interface to URB

VFE sends the assembled root thread payload to URB via a wide data bus. In Generic mode, the data comes from the command as inline or indirect data objects. In IS mode, the inline data is directly assembled as URB register wide payloads, and the indirect data are assembled through the Inverse Scan logic. In VLD mode, the data is decoded from the indirect object (i.e. bitstream data).

10.4.2 Mode of Operations

10.4.2.1 Generic Mode

In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. As there is no special fixed function logic used, the Generic mode can also be viewed as a 'pass-through' mode. In this mode, VFE generates a new thread for each MEDIA_OBJECT command. The payload contained in the MEDIA_OBJECT command (inline and/or indirect) is streamed into URB. The interface descriptor pointer is computed by VFE based on the interface descriptor offset value and the interface descriptor base pointer stored in the VFE state. VFE then forwards the interface descriptor pointer and the URB handle to TS to generate a new root thread. Many media processing applications can be supported using the Generic mode: MPEG-2 HWMC, frame rate conversion, advanced deinterface filter, to name a few.

10.4.2.1.1 Interface Descriptor Selection

After populating the URB with the data, VFE notifies TS to initiate the thread. TS needs an interface descriptor pointer to fetch the information for thread initiation. A list of interface descriptors is arranged by the host software as a descriptor array in memory, as shown in the media state model in Figure 10-8.

VFE obtains the interface descriptor base pointer from the VFE state structure. The offset into the list of interface descriptors comes from MEDIA_OBJECT command. Each interface descriptor has a fixed size. VFE uses a multiple of the fixed size and the offset to add to the base pointer, and creates the final interface descriptor pointer to be sent to TS.

TS fetches the interface descriptor through the Instruction State Cache (ISC) using the interface descriptor pointer. TS then initializes the thread through the Thread Dispatcher. The interface descriptor pointer is given to TS by VFE for a root thread and by a thread for a child thread. The R0 header is formed by TS for a root thread and is stored in URB by the parent thread for a child thread.

10.4.2.1.2 Scratch Space Allocation

TS handles the allocation of scratch space. Since TS does not have a normal state interface, VFE receives the scratch space configuration with the VFE state, then forwards the configuration to TS with the interface descriptor pointer.

10.4.2.2 IS Mode

In Inverse Scan (IS) mode, the Inverse Scan unit, designed to support MPEG-2 standard, is used. In particular, GEN4 architecture can be used to support off-host IDCT acceleration for MPEG-2.

In this mode, a new thread is generated for each MEDIA_OBJECT command. One MEDIA_OBJECT command corresponds to a macroblock. The indirect payload in the command contains the non-zero DCT coefficients for all coded blocks in the macroblock. Detailed data format can be found in section 10.7.2.2.

The indirect payload is streamed into the IS unit. IS unit process the non-zero DCT coefficients on a block-by-block basis. The 16-bit non-zero coefficients are placed in its location within an 8x8 block according to the (x,y) addresses. Hardware fills the rest of the coefficients in the block to zero. The assembled DCT data blocks are then written into URB.

Note that the index for a non-zero coefficient is in row-major order (x, y) address. Host software is responsible of converting the coding scan order to this unified row-major order (e.g. zig-zag scan or alternative scan order such as vertical scan found in MPEG-2 or other coding standard).

Blocks that are not coded will not have coefficient data in the message to the kernel, and the coded blocks are packed back to back. As the message size is variable, VFE calculates the final message size according to the coded block pattern field before sending it to TS.

Interface descriptor select and scratch memory allocation are handled in the same way as in Generic mode.

10.4.2.3 VLD Mode

In VLD mode, both the VLD unit and the IS unit in VFE are used. The VLD unit is specifically designed to support MPEG-2 variable length decoding. Each MEDIA_OBJECT command contains compressed bitstream data associated with a slice. A slice, according to MPEG-2 compressed video bitstream syntax show in Figure 10-13, is the smallest unit that marked by byte-aligned start-code, allowing easy parsing by host software. A slice contains one or more macroblocks. Unlike the other two modes, one or many threads may be generated for each MEDIA_OBJECT command. Each thread corresponds to a macroblock. The indirect payload in the MEDIA_OBJECT command contains the bitstream data for a slice. The indirect payload is streamed into the VLD unit. The decoded non-zero coefficients are then sent to the IS unit. And then the IDCT data blocks (8x8 size) output from the IS unit are then written into URB. For each macroblock, VFE generates the interface descriptor pointer based the decoded macroblock type.

VFE partially decodes (VLD and IS) the MPEG-2 bitstream for a slice and assembles resulting data on a macroblock by macroblock basis for threads running on EUs to

complete the rest of the work. The macroblock-based thread (referred to as a post-VLD thread hereafter) performs inverse quantization, inverse DCT, and motion compensation in order to generate the final output picture. VFE also handles skipped macroblocks so that each post-VLD thread operates on one and only one macroblock.

For bitstreams that are MPEG-2 standard compliant, the output from the VFE fixed function hardware is bit accurate. Bit precision difference may be caused by the IDCT implementation in the kernel. The IDCT kernel must meet the IEEE standard requirement for IDCT.

For bitstreams that are not MPEG-2 standard compliant due to, for example, data corruption, output from VFE fixed function may be unpredictable. That may result in data corruption in the destination buffer after kernel operation. However, VFE fixed function will continue functioning (without hanging).

VFE decodes the slice through the following three major stages: variable length decode, inverse scan and output formatting.

10.4.2.3.1 Variable Length Decode

Variable Length Decode (VLD) stage contains the following sub-stages: data parser, symbol decoder, and motion vector (MV) predictor.

Data Parser

Slice data are processed a dword at a time. Using the byte offset and bit offset provided by the MEDIA_OBJECT command, data parser determines the start bit and sends the slice data to the decoding stage.

Data parser tracks the length of the slice, which is provided by the MEDIA_OBJECT command. Data parser uses the slice length and the starting offsets to calculate the end of slice. When the end of slice is reached, data parser indicates end of slice to symbol decoder and does not pass on any more data that comes from the command stream until a new slice begins.

Symbol Decoder

Symbol decoder performs variable length decoding of the slice bitstream according to the MPEG-2 standard. The decoder analyzes symbols in the bitstream and separates them for further processing. For example, motion vector differentials are sent to motion vector predictor but DCT coefficients are sent directly to IS stage.

Motion Vector Predictor

Motion Vector (MV) Predictor calculates the motion vectors based on the motion vector differentials received from symbol decoder and the motion vector prediction values maintained within MV Predictor, updates the motion vector prediction values accordingly and performs additional arithmetic for dual prime motion vectors to convert them to uni/bi-directional motion vectors. The output motion vectors are relative to the current macroblock position.

10.4.2.3.2 Inverse Scan

IS unit process the non-zero DCT coefficients with their (x, y) location within an 8x8 block received from VLD on a block by block basis. For each new block of data, IS initializes the 8x8 block storage to zero. For each non-zero coefficient received from VLD, IS first sign-extend it to a 16-bit signed value and then place it in the block storage at the location identified by its (x, y) address. When the end of block signal is received from VLD, IS writes the assembled DCT data block into URB.

Only the coded blocks are assembled in the URB, and they are assembled back to back. As the thread payload size is variable, VFE calculates the final message size according to the coded block pattern field before sending the payload size to TS.

10.4.2.3.3 Output Formatting

Additional functionality after inverse scan formats the data that is sent as thread payload to the kernel. Some of this functionality, such as expansion of skip macroblocks and determination of second P field, is done by hardware to make the kernel more efficient.

Skip Macroblocks

VFE processes skip macroblocks by separating them into individual macroblocks and forming one thread for one macroblock. For each skip macroblock, hardware sets its coded block pattern to 0, indicating that no error data is present. All contiguous skip macroblocks have the same relative motion vector. Hardware also handles the difference of skipped macroblocks in a P picture or a B picture as defined by MPEG-2 specification. According to MPEG-2 specification, skip macroblocks cannot extend beyond the end of the current line.

Second Field

According to MPEG-2 specification, field prediction for a P field picture uses the most recently decoded two fields as reference, namely, the most recently decoded reference top field and the most recently decoded reference bottom field. As shown in Figure 10-3, when the current P field is the first field of a frame, both of its reference fields come from the same frame. When the current P field is the second field of a frame, one of its reference fields comes from the same frame. This is illustrated in Figure 10-4.

Detecting second field is important if reference frame selection is required. This is no longer true for GEN4 as each reference field is specified by unique binding table index. Each binding table index contains the pointer to the surface state, which contains not only the field indication but also the base address of the frame buffer. Therefore, it is up to the kernel developer to determine whether to use the Second Field information provided by the hardware.

VFE sets the second field indicator under the following conditions:

- Picture coding type is P picture
- Destination format is field
- Motion type may be field, 16x8 or dual prime
- Either:
  - Top field is first, and
  - Current field is field 1, and

  OR
  - Top field is not first, and
  - Current field is field 0, and

Figure 10-3. Prediction for a P field picture that is a first field, which is (a) a top field, or (b) a bottom field.

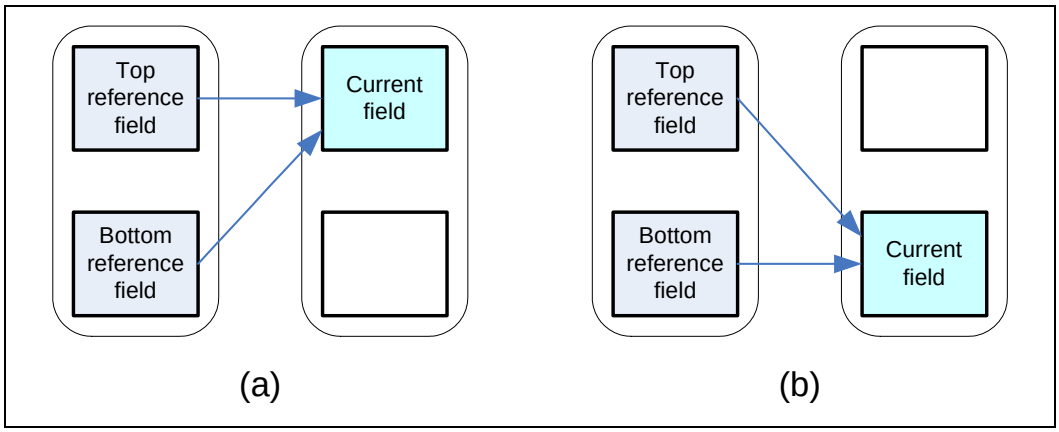
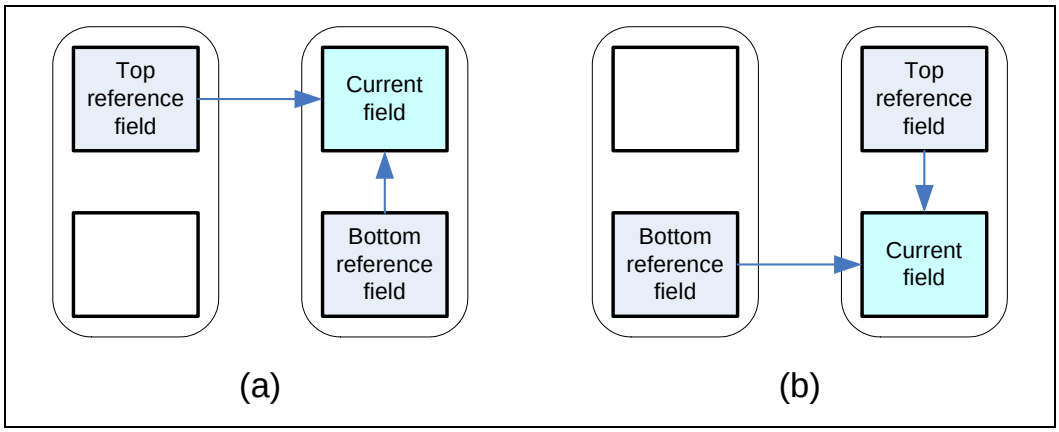


Figure 10-4. Prediction for a P field picture that is a second field, which is (a) a top field, or (b) a bottom field



10.4.2.3.4 Handling Motion Vectors

Table 10-1 provides a summary of different motion types and associated properties. For Frame_Motion_Type, there are three types of Prediction_Type: frame-based prediction, field-based prediction and dual-prime prediction. For Field_Motion_Type, there are three types of Prediction_Type: field-based prediction, dual-prime prediction and 16x8 prediction.

Table 10-2 details the motion compensation operations for various frame motion types and Table 10-3 depicts the motion compensation operations for various field motion types.

Table 10-1. Summary of Motion Types

*_Motion_Type	Prediction_Type	Vector [r][s]	Possible MV Combinations in Bitstream	Uses Motion Vertical Field Select
Frame	Frame-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	None,0,1,0+1	No
Frame	Field-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	0+2,1+3, 0+1+2+3	Yes
Frame	Dual-Prime	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3 [2][0] – 4 [3][0] – 6	0+4+6 (See Frame-Dual Prime table)	No
Field	Field-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	None, 0, 1, 0+1	Yes
Field	Dual-Prime	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3 [2][0] – 4	0+4 (See Field-Dual Prime table)	Yes
Field	16x8	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	0+2, 1+3,0+1+2+3	Yes

*Vectors 4 and 6 are the derived motion vectors (DMVs) for dual-prime prediction that are calculated by PR and placed in the thread payload in the specified MVector position.

Table 10-2. Motion Comp Operation for Pictures with Frame Motion Type

frame_motion_type	forward	backward	intra	Motion vector (v'[r][s][t])	Command	HW Mvector (MV[r][s])	**Prediction Map**	Prediction formed for	MVFS
Frame-based‡	-	-	1	v'[0][0][1:0]	0	-	-	None (motion vector is for concealment)	-
Frame-based	1	1	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	frame, forward	-
				v'[0][1][1:0]		MV[0][1]	Back	frame, backward	-
Frame-based	1	0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	frame, forward	-
Frame-based	0	1	0	v'[0][1][1:0]	2	MV[0][1]	Back	frame, backward	-
Frame-based‡	0 (1)	0	0	v'[0][0][1:0]*§	2	MV[0][1]	Fwd	frame, forward	-
Field-based	1	1	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, forward	[0][0]
				v'[1][0][1:0]		MV[1][0]	Fwd	bottom field, forward	[1][0]
				v'[0][1][1:0]		MV[0][1]	Back	top field, backward	[0][1]
				v'[1][1][1:0]		MV[1][1]	Back	bottom field, backward	[1][1]
Field-based	1	0	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, forward	[0][0]
				v'[1][0][1:0]		MV[1][0]	Fwd	bottom field, forward	[1][0]
Field-based	0	1	0	v'[0][1][1:0]	4	MV[0][1]	Back	top field, backward	[0][1]
				v'[1][1][1:0]		MV[1][1]	Back	bottom field, backward	[1][1]
Dual prime	1	0 (1)	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, from same parity, forward	[0][0] = 0
				v'[0][0][1:0]		MV[1][0]	Fwd	bottom field, from same parity, forward	[1][0] = 1
				v'[2][0][1:0]*†		MV[0][1]	Fwd	top field, from opposite parity, forward	[0][1] = 1
				v'[3][0][1:0]*†		MV[1][1]	Fwd	bottom field, from opposite parity, forward	[1][1] = 0

NOTE -	Motion vectors are listed in the order they appear in the bitstream
?	the motion vector is only present if concealment_motion_vectors is one
‡	frame_motion_type is not present in the bitstream but is assumed to be Frame-based
*	These motion vectors are not present in the bitstream
†	These motion vectors are derived from vector'[0][0][1:0] as described in 7.6.3.6
§	The motion vector is taken to be (0; 0) as explained in 7.6.3.5

Table 10-3. Motion Comp Operation with Field Motion Type

					Prediction Map (SecondPField|BottomField)					
backward	intra	Motion vector	Command	HW MVector	00	01	10	11	Prediction formed for	
-	1	v'[0][0][1:0]?		None	N/A	-	-	-	None (motion vector is for concealment)	
1	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	Fwd	x	x	whole field, forward	B-Pict only
		v'[0][1][1:0]		MV[0][1]	Back	Back	x	x	whole field, backward	
0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) if MVFS[0][0]=0	Dst (M1) if MVFS[0][0]=0	whole field, forward	
							Dst (M1) if MVFS[0][0]=1	Fwd (M0) if MVFS[0][0]=1		
0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	Fwd	x	x	whole field, forward	
1	0	v'[0][1][1:0]	2	MV[0][1]	Back	Back	x	x	whole field, backward	B-Pict only
0	0	v'[0][0][1:0]*§	**2**	MV[0][0]	Fwd	Fwd	Fwd w/ MVFS[0][0]=0	Fwd w/ MVFS[0][0]=1	whole field, forward	P-Pict only, Same parity.
1	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	Fwd	x	x	upper 16x8 field, forward	B-Pict only
		v'[1][0][1:0]		MV[1][0]	Fwd	Fwd	x	x	lower 16x8 field, forward	
		v'[0][1][1:0]		MV[0][1]	Back	Back	x	x	upper 16x8 field, backward	
		v'[1][1][1:0]		MV[1][1]	Back	Back	x	x	lower 16x8 field, backward	
0	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	Fwd	Fwd (M0) if MVFS[0][0]=0	Dst (M1) if MVFS[0][0]=0	upper 16x8 field, forward	
							Dst (M1) if MVFS[0][0]=1	Fwd (M0) if MVFS[0][0]=1		
		v'[1][0][1:0]		MV[1][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) if MVFS[1][0]=0	Dst (M1) if MVFS[1][0]=0	lower 16x8 field, forward	
							Dst (M1) if MVFS[1][0]=1	Fwd (M0) if MVFS[1][0]=1		
1	0	v'[0][1][1:0]	4	MV[0][1]	Back	Back	x	x	upper 16x8 field, backward	B-Pict only
		v'[1][1][1:0]		MV[1][1]	Back	Back	x	x	lower 16x8 field, backward	
0 (1) %	0	v'[0][0][1:0]	2	MV[0][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) w/ MVFS[0][0]=0	Fwd (M0) w/ MVFS[0][0]=1	whole field, from same parity, forward	P-Pict only (SW forces MVFS[0][0], MVFS[0][1])
		v'[2][0][1:0]*†		MV[0][1]	Fwd (M1)	Fwd (M1)	Dest (M1) w/ MVFS[0][1]=1	Dest (M1) w/ MVFS[0][1]=0	whole field, from opposite parity, forward	

Notes: Motion vectors are listed in the order they appear in the bitstream.

? — The motion vector is only present if concealment_motion_vectors is one.

‡ — Field_motion_type is not present in the bitstream but is assumed to be Field-based.

* — These motion vectors are not present in the bitstream.

† — These motion vectors are derived from vector'[0][0][1:0] as described in 7.6.3.6.

§ — The motion vector is taken to be (0; 0) as explained in 7.6.3.5.

% — Software converts the motion type. For Dual-prime case, software converts it to a bidirectional prediction.

Remarks:

- Forward, Backward MV pairs always used for combined prediction (Bi-dir or Dual-Prime)
- MVs [0][0] and [0][1] can be to Fwd reference, Backward reference, or Destinsation buffer (for 2nd field case)
- MV [1][0] can be to a Fwd or Dest
- MV [1][1] is always to a Back
- (M0), (M1) stands for MIP_INFO setting for the first reference picture and the second reference picture. It is particularly important to set correct M1 for P-pictures to deal with SecondField and DualPrime cases.
- Software converts the dual-prime case to a field-based bidirectional prediction with 2 MVs.

10.4.2.3.5 Dual Prime Handling

Dual prime prediction is only valid for a P-picture. In dual prime mode, each field will have two predictions similar to the forward and backward predictions in a B-picture, as the final prediction value for the field is the average of the two. One of the motion vectors is provided by the bitstream and the other one is derived. Motion Vector Predictor unit is responsible for converting all dual prime predictions to a forward and backward field prediction according the Table 10-4 for P frame picture and Table 10-5 for P field picture.

Table 10-4. Converting Frame-Dual Prime Motion to 4MV

Prediction formed for: Field / Parity	MPEG-2 MV[r][s]	Thread Payload MV[r][s]	Motion Vertical Field Select (MVFS)	
Top / Same	[0][0]	[0][0]	Bit 0 = 0	MVFS = 6h
Bottom / Same	[0][0]	[1][0]	Bit 2 = 1	
Not Used	[0][1]	-		
Not Used	[1][0]	-		
Not Used	[1][1]	-		
Top / Opposite	[2][0]	[0][1]	Bit 1 = 1	
Bottom / Opposite	[3][0]	[1][1]	Bit 3 = 0	

Table 10-5. Converting Field-Dual Prime Motion to 2MV

Prediction formed for: Field / Parity	MPEG-2 MV[r][s]	Thread Payload MV[r][s]	MotionVerticalFieldSelect	
			Top Field	Bottom Field
Whole field / Same	[0][0]	[0][0]	Bit 0 = 0	Bit 0 = 1
Whole field / Opposite	[2][0]	[0][1]	Bit 1 = 1	Bit 1 = 0

10.4.2.3.6 Interface Descriptor Selection

In VLD mode, the Interface Descriptor Offset field in the MEDIA_OBJECT command is ignored by hardware. Instead, the interface descriptor offset is computed by hardware based on the decoded macroblock parameters and a remapping table.

First a macroblock index is computed based on parameters such as picture structure, motion type, prediction type, DCT type, intra-coding type and motion vector present information. Table 10-6 provides the macroblock index table for a frame-picture destination buffer (with Picture Structure = 11). Table 10-7 shows macroblock indices for a field-picture destination buffer (with Picture Structure = 01 or 10). As Picture Structure is a state variable that will not be changed until a pipeline flush, the macroblock indices can be computed separately for different Picture Structure.

After the macroblock index is computed, it is used as the index into the Interface Descriptor Remap Table to derive the final interface descriptor offset value. The Interface Descriptor Remap Table is provided as part of the VLD state.

The interface descriptor offset value multiplied by the interface descriptor size is then added to the interface descriptor base pointer to generate the interface descriptor pointer for the post-VLD thread.

The last three columns in Table 10-6 and Table 10-7 indicate whether a macroblock index is applicable for a given Picture Coding type (I, P or B). A ‘Y’ (or a ‘N’) means the macroblock index on the row is valid (or invalid) for the Picture Type shown on the column. Taking a frame picture destination for example, only macroblock indices 0 and 8 are valid for an I-picture; indices 0-3 and 8-11 are valid for a P-picture; and for a B-picture, only indices 3 and 11 are not valid.

Developers can use the remap table for kernel development to fine-tune system performance and reduce software complexity. For example, if the destination is a frame picture, the kernel for a macroblock with dual-prime motion in a P-picture (macroblock index = 3) may be identical to that for a macroblock with bidirection field motion in a B picture (macroblock index = 7). A common set of interface descriptors can be configured once for frame picture destinations, and reused without change when the destination is of I-, P- and B- picture coding type.

In another case, if it is determined that kernel software will be responsible of handling DCT types for a frame picture destination, then macroblock index i and i+8 , for i = 0 to 7, can be mapped to the same interface descriptor.

Table 10-6. Macroblock indices for frame picture destination

Macroblock Index	Interface Descriptor Kernel Function (Frame Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward frame motion	N	Y	Y
2	Forward field motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward frame motion	N	N	Y
5	Backward field motion	N	N	Y
6	Bidirectional frame motion	N	N	Y
7	Bidirectional field motion	N	N	Y
8	I macroblock w/ field DCT	Y	Y	Y
9	Forward frame motion w/ field DCT	N	Y	Y
10	Forward field motion w/ field DCT	N	Y	Y
11	P picture, dual-prime motion w/ field DCT	N	Y	N
12	Backward frame motion w/ field DCT	N	N	Y
13	Backward field motion w/ field DCT	N	N	Y
14	Bidirectional frame motion w/ field DCT	N	N	Y
15	Bidirectional field motion w/ field DCT	N	N	Y

Table 10-7. Macroblock indices for field picture destination

Macroblock Index	Interface Descriptor Kernel Function (Field Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward field motion	N	Y	Y
2	Forward 16x8 motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward field motion	N	N	Y
5	Backward 16x8 motion	N	N	Y
6	Bidirectional field motion	N	N	Y
7	Bidirectional 16x8 motion	N	N	Y

10.4.3 Debug Counter

VFE contains a counter for software debug. This is a 24-bit free running counter that increments each time a new root thread is delivered from VFE to TS, regardless whether the thread is passed to VFE from Command Streamer (as in Generic mode or IS mode) or created by VFE (as in VLD mode).

Software can choose to reset (initialize to the object ID) this free-running counter upon a new primitive command or upon a state change. Software can also choose to leave the Debug Counter value unchanged (frozen).

The Debug Counter is initialized to 0 after power up and hardware/software reset.

The following table shows the usage of the state fields controlling the debug counter – Debug Counter Control field in VFE state.

Debug Counter Control	In Generic Mode or IS Mode	In VLD Mode
00	Free Running Counter: Debug counter maintains its value from the previous VFE state and increment by 1 for each new MEDIA_OBJECT command. It is used as a free running root thread counter.	Free Running Counter: Debug counter maintains its value from the previous state and increment by 1 for each new macroblock thread generated by the VLD/IS hardware functions (e.g. may increment by many for a MEDIA_OBJECT command). As the media pipeline may be used for different applications within the same context, the debug counter is treated here as a free-running counter.
01	**Frozen Counter:** Debug counter maintains its value from the previous VFE state and is unchanged by subsequent MEDIA_OBJECT commands. It may be used by a context that is not debugged.	
10	Initialized Once: Debug counter is initialized to the Object_ID number in the first MEDIA_OBJECT command after the media pipeline state change. It is then incremented by one for each subsequent MEDIA_OBJECT command. Properly setting the OBJECT_ID field for the first MEDIA_OBJECT command allows uniquely tracking of all the threads in groups based on state changes.	Initialized Once: Debug counter is reset to the Object_ID number for the first macroblock of the first slice after the media pipeline state change and then increment for subsequent macroblocks in the slice and for these in subsequent slices. Assuming VFE state change is at the beginning of each frame, the debug counter is then the macroblock count for a video frame. When new OBJECT_ID is used for each frame, the debug counter for a thread can be used to uniquely determine a macroblock in a decoding video frame sequence.

Debug Counter Control	In Generic Mode or IS Mode	In VLD Mode
11	Always Initialized: Debug counter is reset to the Object_ID number for every MEDIA_OBJECT command. Host software is in full control of the debug counter field – unique number should be assigned to the Object_ID field for each command.	Reserved. (This configuration is not allowed in VLD mode. In VLD mode, the Debug counter can only be reset after state change.)

Debug Counter Control	In AVC-IT, AVC-MC or VC1-IT mode
00	Free Running Counter: Debug counter maintains its value from the previous VFE state and increment by 1 for each new media primitive command. It is used as a free running root thread counter.
01	**Frozen Counter:** Debug counter maintains its value from the previous VFE state and is unchanged by subsequent media primitive commands. It may be used by a context that is not debugged.
10	Initialized Once: Debug counter is initialized to the Object_ID number in the state field in VFE_STATE_EX after the media pipeline state change. It is then incremented by one for each subsequent media primitive command. Properly setting the OBJECT_ID field in the VFE_STATE_EX field allows uniquely tracking of all the threads in groups based on state changes.
11	Reserved. (The Debug counter can only be reset after state change.)

The Debug Counter is saved/restored during context switch.

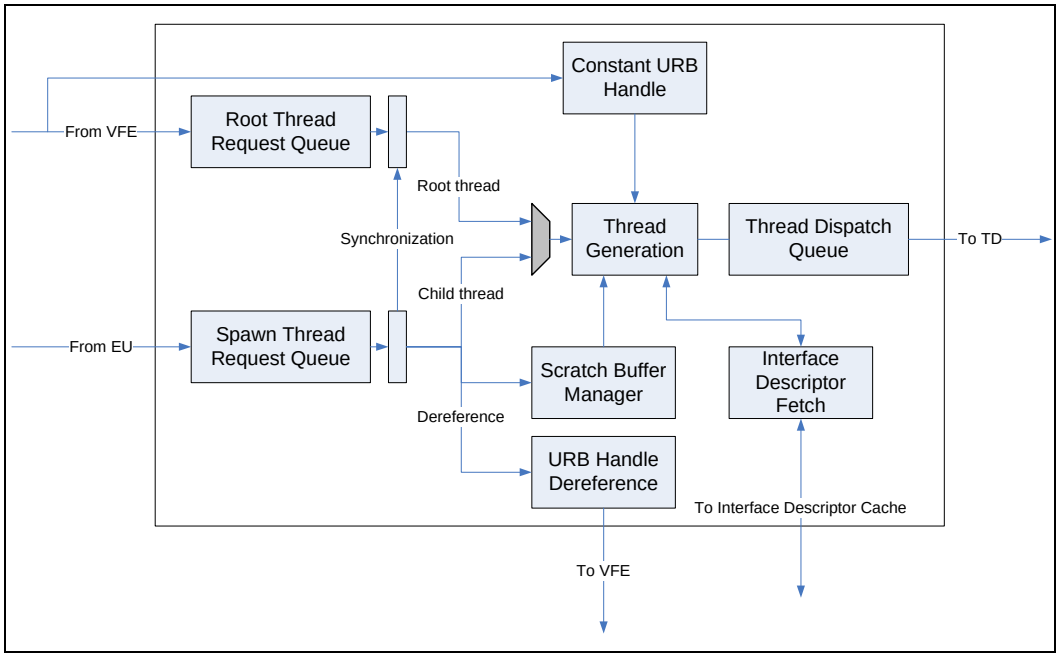
The Debug Counter is not saved/restored during context switch.

The Debug Counter is not saved/restored during context switch. Therefore, when the media pipe is used by multiple contexts, debug counter may only be used by a single context. In order to not interfere the context being debugged, the debug counter should be left unchanged by software in other contexts.

10.5 Thread Spawner Unit

The Thread Spawner (TS) unit is responsible for making thread requests (root and child) to the Thread Dispatcher, managing scratch memory, maintaining outstanding root thread counts, and monitoring the termination of threads.

Figure 10-5. Thread Spawner block diagram



10.5.1 Basic Functions

10.5.1.1 Root Threads Lifecycle

Thread requests sourced from VFE are called **root threads**, since these threads may be creating subsequent (child) threads. A root thread may a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode.

Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.

Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
   - Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.
2. TS then builds the transparent header and the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.
   - It is the root thread's responsibility (software) to guarantee that all its children have retired before the root thread can retire.

10.5.1.2 URB Handles

VFE is in charge of allocating URB handles for root threads. One URB handle is assigned to each root thread. The handle is used for the payload into the root thread.

If Children Present state variable is not set (root-without-child mode), TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched.

If Children Present state variable is set (root-with-child mode), the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.

10.5.1.3 Root to Child Responsibilities

Any thread created by another thread running in an EU is called a **child thread**. Child threads can create additional threads, all under the tree of a root which was requested via the VFE path.

A root thread is responsible of managing pre-allocated resources such as URB space and scratch space for its direct and indirect child threads. For example, a root thread may split its URB space into sections. It can use one section for delivering payload to one child thread as well as forwarding the section to the child thread to be used as return URB space. The child thread may further subdivide the URB section into subsections and use these subsections for its own child threads. Such process may be iterated. Similarly, a root thread may split its scratch memory space into sections and give one scratch section for one child thread.

TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

10.5.1.4 Multiple Simultaneous Roots

Multiple root threads are allowed concurrently running in GEN4 execution units. As there is only one scratch space state variable shared for all root threads, all concurrent root thread requiring scratch space share the same scratch memory size. Figure 10-6 depicts two examples of thread-thread relationship. The left graph shows one single tree structure. This tree starts with a single root thread that generates many child threads. Some child threads may create subsequent child threads. The right graph shows a case with multiple disconnected trees. It has multiple root threads, showing sibling roots of disconnected trees. Some roots may have child threads (branches and leafs) and some may not.

There is another case (as shown in Figure 10-7) where multiple trees may be connected. If a root is a synchronized root thread, it may be dependent on a preceding sibling root thread or on a child thread.

Figure 10-6. Examples of thread relationship

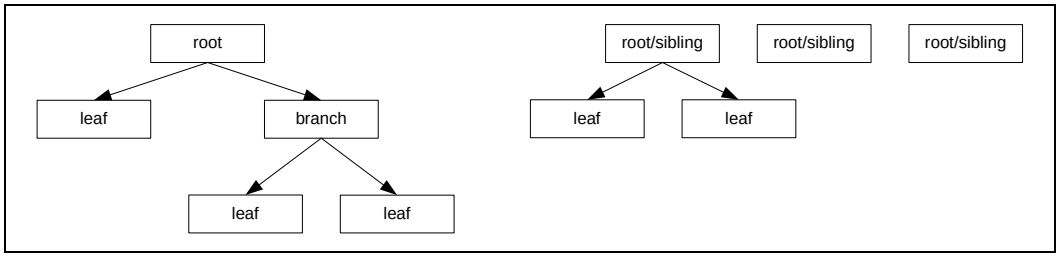
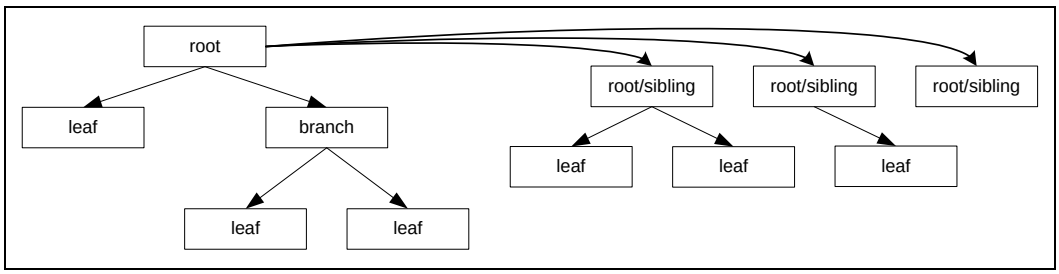


Figure 10-7. An example of thread relationship with root sibling dependency



10.5.1.5 Synchronized Root Threads

A synchronized root thread (SRT) originates from a MEDIA_OBJECT command with Thread Synchronization field set. Synchronized root threads share the same root thread request queue with the non-synchronized roots. A SRT is not automatically dispatched. Instead, it stays in the root thread request queue until a spawn-root message is at the head of the child thread request queue. Conversely, a spawn-root message in the child thread request queue will block the child thread request queue until the head of root thread request queue is a SRT. When they are both at the head of queues, they are taken out from the queue at the same time.

A spawn-root message may be issued by a root thread or a child thread. There is no restriction. However, the number of spawn-root messages and the number of SRT must be identical between state changes. Otherwise, there can be a deadlock. Furthermore, as both requests are blocking, synchronized root threads must be used carefully to avoid deadlock.

When Scoreboard Control is enabled, the dispatch of a SRT originated from a MEDIA_OBJECT_EX command is still managed by the same way in addition to the hardware scoreboard control.

10.5.1.6 Deadlock Prevention

Root threads must control deadlock within their own child set. Each root is given a set of preallocated URB space; to prevent deadlock it must make sure that all the URB space is not allocated to intermediate children who must create more children before they can exit.

There are limits to the number of concurrent threads. The upper bound is determined by the number of execution units and the number of threads per EU. The actual upper bound on number of concurrent threads may be smaller if the GRF requirement is large. Deadlock may occur if a root or intermediate parent cannot exit until it has started its children but there is no space (for example, available thread slot in execution units) for its children to start.

To prevent deadlock, the maximum number of root threads is provided in VFE state. The Thread Spawner keeps track of how many roots have been spawned and prevents new roots if the maximum has been reached. When child threads are present, it is software's responsible of constraining child thread generation, particularly the generation of child threads that may also spawn more child threads.

Child thread dispatch queue in TS is another resource that needs to be considered in preventing deadlock. The child thread dispatch queue in TS is used for (1) message to spawn a child thread, (2) message to spawn a synchronized root thread, and (3) thread termination message. If this queue is full, it will prevent any thread to terminate, causing deadlock.

For example, if an application only has one root thread (max # of root threads is programmed to be one). This root thread spawns child threads. In order to avoid deadlock, the maximum number of outstanding child thread that this root thread can spawn is the sum of the maximum available thread slots plus the depth of the child thread dispatch queue minus one.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1)

Adding other root threads (synchronized and/or non-synchronized) to the above example, the situation is more complicated. A conservative measure may have to use to prevent deadlock. For example, the root thread spawning child threads may have to exclude the max number of root threads as in the following equation to compute the maximum number of outstanding child threads to be dispatched.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1) – (Max Root Threads-1)

Table 10-8. TS Resource Available in Device Hardware

Device	Child Thread Dispatch Queue Depth
[DevBW]	8
[DevCL]	8

10.5.1.7 Child Thread Lifecycle

When a (parent) thread creates a child thread, the parent thread behaves like a fixed function. It provides all necessary information to start the child thread, by assembling the payload in URB (including R0 header) and then sending a spawn thread message to TS with following data:

- An interface descriptor pointer for the child thread.
- A pointer for URB data

The interface descriptor for a child may be different from the parent – how the parent determines the child interface descriptor is up to the parent, but it must be one from the interface descriptor array on the same interface descriptor base address.

The URB pointer is not the same as a URB handle. It does not have an URB handle number and does not appear in any handle table. This is acceptable because the URB space is never reclaimed by TS after a child is dispatched, but rather when the parent releases its original handles and/or retires.

The R0 header for a child, as part of the URB payload, also includes debug fields for the child, consisting of the 32-bit field from the parent and a parent created field to uniquely identify the child.

The child request is stored in the child thread queue. The depth of the queue is limited to 8, overrun is prevented by the message bus arbiter which controls the message bus. The arbiter knows the depth of the queue and will only allow 8 requests to be outstanding until the TS signals an entry has been removed.

As mentioned previously, child threads have higher priority over root threads. Once TS selects a child thread to dispatch, it follows these steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
2. TS then builds the transparent header but not the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. Once the dispatch is done, TS can forget the child – unlike roots, no bookkeeping is done that has to be updated when the child retires.

If more data needs to be transferred between a parent thread and its child thread than that can fit in a single URB payload, extra data must be communicated via shared memory through data port.

10.5.1.8 Arbitration between Root and Child Threads

When both root thread queue and child thread queue are both non-empty, TS serves the child thread queue. In other words, child threads have higher priority over root threads. The only condition that the child thread queue is stalled by the root thread queue is that the head of child thread queue is a root-synchronization message and the head of root thread queue is not a synchronized root thread.

10.5.2 Interfaces

10.5.2.1 Interface to VFE

TS receives an interface descriptor pointer and a URB handle from VFE. It uses the interface descriptor pointer to fetch the interface descriptor. TS uses the information in the interface descriptor along with the URB handle to fill out the transparent header in the message to TD for all threads. For root thread, TS also generate the R0 header.

TS transmits URB handle dereference signal to VFE. As described previously, the derefernce signal may be at dispatch time or at later time depending on Children Present state variable. No matter which case, there is one and only one URB handle dereference for a thread.

10.5.2.2 Interface to Thread Dispatcher

TS creates the transparent header, assembles the URB handles and calls TD to dispatch a new thread. For an unsynchronized root thread, there is one URB handle managed by VFE and optionally one Constant URB handle managed by CS. For a synchronized root thread, there is one URB handle managed by VFE, a URB handle created by the synchronizing thread (the one that sends the ‘spawn root thread’ message, and optionally one Constant URB handle managed by CS. For a child thread, there is one URB handle managed by the parent thread plus an optional Constant URB handle.

10.6 Media State

10.6.1 Media State Model

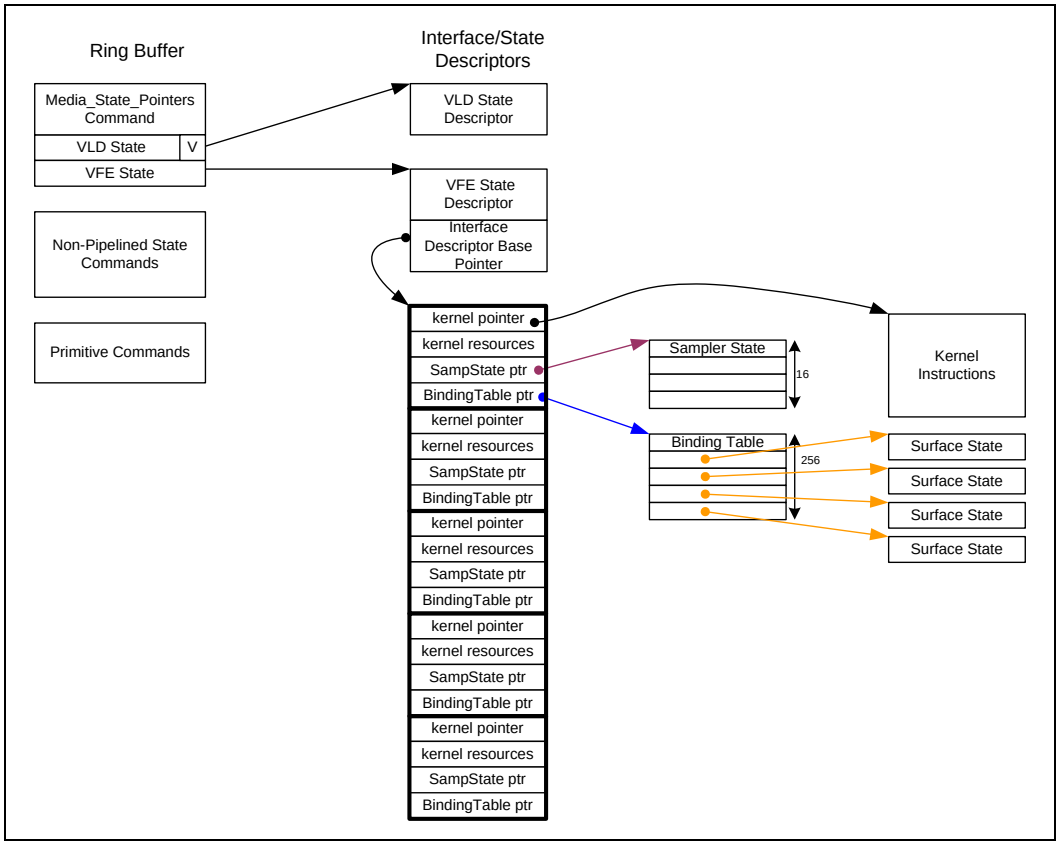
The media state model is based on an indirect state fetching mechanism. State Descriptors provide state information for fixed function units of the media pipeline. Interface Descriptors provide state information for kernels (threads) dispatched from the media pipeline. There are organized in different memory locations.

VFE State Descriptor contains states for both VFE unit and TS unit. The special purpose VLD state information is provided by a separate VLD State Descriptor.

All Interface Descriptors have the same size and are organized as a contiguous array in memory. They can be selected by Interface Descriptor Index for a given kernel. This allows different kinds of kernels to coexist in the system.

The MEDIA_STATE_POINTERS command provides the memory pointers to the Descriptors.

Figure 10-8. Media State Model



10.6.2 VFE_STATE

Dword	Bit	Description
0	31:10	**Scratch Space Base Pointer.** Specifies the 1k-byte aligned address offset to scratch space for use by the kernel. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Reserved: MBZ
	7	**Extended VFE State Present.** This field specifies whether extended VFE state is present or not. It must be programmed with the same value as the **Extended VFE State Enable** field in MEDIA_STATE_POINTERs command. 0 = Disabled. No extended VFE state (and Extension State Pointer is ignored). 1 = Enabled. The extended VFE state pointed by Extended State Pointer is loaded [**DevBW, DevCL**] This field is reserved and MBZ.
	6:4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space allowed to be used by each thread. The driver must allocate enough contiguous scratch space, pointed to by the Scratch Space Pointer, to ensure that the Maximum Number of Threads each get Per Thread Scratch Space size without exceeding the driver-allocated scratch space. **Note:** The definition of this field is different from that in 3D fixed functions, where the per-thread scratch space is specified in powers of 2. Format = U4 Range = [0,11] indicating [1k bytes, 12k bytes]
1	31:25	**Maximum Number of Threads.** Specifies the maximum number of simultaneous root threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. Note that MSB will be zero due to the range limit below. Format = U7 representing (thread count – 1) Range = [0, n-1] where n = (# EUs) * (# threads/EU). See *Graphics Processing Engine* for listing of #EUs and #threads in each device.
	24:16	**URB Entry Allocation Size.** Specifies the length of each URB entry used by the unit, in 512-bit register increments - 1. Format = U9 Range = [0,255] indicating [1,256] 512-bit register increments
	15:9	**Number of URB Entries.** Specifies the number of URB entries that are used by the unit. Format = U7 Range = [1,64]
	8:7	Reserved : MBZ

Dword	Bit	Description
	6:3	**VFE Mode** 0000 – **Generic** Mode 0001 – **VLD** Mode (MPEG-2 only) 0010 – **IS** Mode 0100 – **AVC-MC** Mode 0111 – **AVC-IT** Mode 1011 – **VC1-IT** Mode All other encodings are reserved [**DevBW, DevCL**] AVC-MC, AVC-IT and VC1-IT modes are not supported [**DevBW**] VLD mode is not supported
	2	**Children Present.** Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. In VLD Mode, this field must be 0. Format = Enable
	1:0	**Debug Counter Control.** This field controls the Debug Counter in VFE. See Section 10.4.3 for more details. 00 – Free Running Debug Counter 01 – Frozen Debug Counter 10 – Debug Counter Initialized Once 11 – Debug Counter Initialized Always (Reserved in VLD mode)
2	31:4	**Interface Descriptor Base Pointer.** Specifies the 16-byte aligned address of the interface descriptor base pointer. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:4]
	3:0	Reserved : MBZ

10.6.3 VLD_STATE

<table>
<tr><th>Dword</th><th>Bit</th><th>Description</th></tr>
<tr><td rowspan="11">0</td><td>31:28</td><td>f_code[1][1]. Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details</td></tr>
<tr><td>27:24</td><td>f_code[1][0]. Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details</td></tr>
<tr><td>23:20</td><td>f_code[0][1]. Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details</td></tr>
<tr><td>19:16</td><td>f_code[0][0]. Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details</td></tr>
<tr><td>15:14</td><td>Intra DC Precision. See ISO/IEC 13818-2 §6.3.10 for details.</td></tr>
<tr><td>13:12</td><td>Picture Structure. This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See ISO/IEC 13818-2 §6.3.10 for details.

Format = MPEG_PICTURE_STRUCTURE
00 = Reserved
01 = MPEG_TOP_FIELD
10 = MPEG_BOTTOM_FIELD
11 = MPEG_FRAME</td></tr>
<tr><td>11</td><td>TFF (Top Field First). When two fields are stored in a picture, this bit indicates if the top field is the first field.

For a frame P picture, the value 1 indicates that the top field of the reconstructed frame is the first field output by the decoding process, the same as defined in ISO/IEC 13818-2 §6.3.10. Particularly, it is used by the hardware to calculate derivative motion vectors from the dual-prime motion vectors.

For a field P picture, hardware uses this bit together with the Picture Structure to determine if the current picture is the Second Field. In this case, the definition of this bit differs from ISO/IEC 13818-2 §6.3.10 – software must derive the value for this bit according to the following relation:
<table>
<tr><th></th><th>Picture Structure = top field</th><th>Picture Structure = bottom field</th></tr>
<tr><td>Second Field = 0</td><td>TFF = 1</td><td>TFF = 0</td></tr>
<tr><td>Second Field = 1</td><td>TFF = 0</td><td>TFF = 1</td></tr>
</table></td></tr>
<tr><td>10</td><td>Frame Prediction Frame DCT. This field provides constraints on the DCT type and prediction type. It affects the syntax of the bitstream.</td></tr>
<tr><td>9</td><td>Concealment Motion Vector Flag. This field indicates if the concealment motion vectors are coded in intra macroblocks. It affects the syntax of the bitstream.</td></tr>
<tr><td>8</td><td>Quantizer Scale Type. This field specifies the quantizer scaling type.

Format = MPEG_Q_SCALE_TYPE
0 = MPEG_QSCALE_LINEAR
1 = MPEG_QSCALE_NONLINEAR</td></tr>
<tr><td>7</td><td>Intra VLC Format. This field is used by VLD.</td></tr>
</table>

Dword	Bit	Description
	6	**Scan Order.** This field specifies the Inverse Scan method for the DCT-domain coefficients in the blocks of the current picture. Format = MPEG_INVERSESCAN_TYPE 0 = MPEG_ZIGZAG_SCAN 1 = MPEG_ALTERNATE_VERTICAL_SCAN
	5:0	**Reserved.**
1	31:14	**Reserved.**
	13	**Reserved (was Concealment Enable)**
	12	**Reserved (was Concealment Reference)**
	11	**Reserved (was Concealment Type)**
	10:9	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	8:1	**Reserved** (was Slice Error Control)
	0	**Reserved.** (was Disable Mismatch)
2	31:0	**Interface Descriptor Remap Table [7:0].** This field contains the interface descriptor remap table entries for the first 8 kernel indices. Each table entry has 4 bits, providing a remapping range of [0, 15]. This field is applicable to both frame picture destination (Picture Structure = 11) and field picture destination (Picture Structure = 01 or 10). Bits 31:28: Remap for index = 7 Bits 27:24: Remap for index = 6 Bits 23:20: Remap for index = 5 Bits 19:16: Remap for index = 4 Bits 15:12: Remap for index = 3 Bits 11:8: Remap for index = 2 Bits 7:4: Remap for index = 1 Bits 3:0: Remap for index = 0 **[DevCL] Errata:** This field is reserved.
3	31:0	**Interface Descriptor Remap Table [15:8].** This field contains the interface descriptor remap table entries for the last 8 kernel indices. Each table entry has 4 bits, providing a remapping range of [0, 15]. This field is only applicable to frame destination. It is ignored when the destination is a field picture. **[DevCL] Errata:** This field is reserved.

10.6.4 INTERFACE_DESCRIPTOR

DWord	Bit	Description
0	31:6	**Kernel Start Pointer.** Specifies the 64-byte aligned address offset of the first instruction in the kernel. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:6]
	5:4	Reserved : MBZ
	3:0	**GRF Register Blocks.** Defines the number of GRF Register Blocks used by the kernel. A register block contains 8 registers. A kernel using a register count that is not a multiple of 8 must round up to the next multiple of 8. Format = U4 register block count - 1 Range = [0,15] corresponding to [1,16] 8-register blocks Restriction: LSB must be zero, indicating that GRF assignment is in granularity of 16 GRF registers.
1	31:26	**Constant URB Entry Read Length.** Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 8-DW register increments. A value 0 means that no Constant URB Entry will be loaded. The Constant URB Entry Read Offset field will then be ignored. Format = U6 Range = [0,63]
	25:20	**Constant URB Entry Read Offset.** Specifies the offset (in 8-DW units) at which Constant URB data is to be read from the URB before being included in the thread payload. Format = U6 Range = [0,63]
	19	Reserved : MBZ
	18	**Single Program Flow (SPF).** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). 0 = Multiple Program Flows 1 = Single Program Flow
	17	**Thread Priority.** Specifies the priority of the thread for dispatch 0 = Normal Priority 1 = High Priority **Programming Notes:** • This field must be set to zero.
	16	**Floating Point Mode.** Specifies the floating point mode used by the dispatched thread. 0 = Use IEEE-754 Rules 1 = Use alternate rules
	15:14	Reserved: MBZ

DWord	Bit	Description
	13	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	12	Reserved: MBZ
	11	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. Format: Enable
	10:8	Reserved: MBZ
	7	**Software Exception Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	6:0	Reserved: MBZ
2	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5] *This field is ignored for child threads.*
	4:2	**Sampler Count.** Specifies how many samplers (in multiples of 4) the kernel uses. Used only for prefetching the associated sampler state entries. Format = U3 Range = [0,4] 0 = no samplers used 1 = between 1 and 4 samplers used 2 = between 5 and 8 samplers used 3 = between 9 and 12 samplers used 4 = between 13 and 16 samplers used *This field is ignored for child threads.* *If this field is not zero, sampler state is prefetched for the first instance of a root thread upon the startup of the media pipeline.*
	1:0	Reserved : MBZ
3	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned address of the binding table. This pointer is relative to the **Surface State Base Address**. Format = SurfaceStateOffset[31:5] *This field is ignored for child threads.*

DWord	Bit	Description
	4:0	**Binding Table Entry Count.** Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** The maximum number of prefetched binding table entries is limited to 31. For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache. Format = U5 Range = [0,31] *This field is ignored for child threads.* *If this field is not zero, binding table and surface state are prefetched for the first instance of a root thread upon the startup of the media pipeline.*

10.7 Media State and Primitive Commands

10.7.1 MEDIA_STATE_POINTERS Command

The MEDIA_STATE_POINTERS command is used to set up the pointers to the VFE states (VFE state, VLD state or VFE state extension). This command is issued prior to a set of media primitive commands, and points to the Generic mode VFE state and VLD decode mode VLD state (or VFE extended state).

[DevBW-A] Errata BWT007: State data pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_STATE_POINTERS Pipeline[28:27] = Media = 2h; Opcode[26:24] = 0h; Subopcode[23:16] **=** 0h
	15:0	**DWord Length** (Excludes DWords 0,1) = 01h
1	31:5	**Extended State Pointer.** Specifies the 32-byte aligned address of the extended VFE state (either VLD_STATE or VFE_STATE_EX). This pointer is relative to the **General State Base Address**. Which extended VFE state is used depends on **VFE Mode**. If **VFE Mode** is set to VLD Mode (0001), VLD_STATE is used. Otherwise, VFE_STATE_EX is used. Format = GeneralStateOffset[31:5] [**DevBW, DevCL**] Note that VFE_STATE_EX is reserved
	4:1	Reserved : MBZ

DWord	Bit	Description
	0	**Extended VFE State Enable (**was **VLD Enable).** This field specifies whether extended VFE state is loaded. 0 = Disabled. No extended VFE state (and Extension State Pointer is ignored). 1 = Enabled. The extended VFE state pointed by Extended State Pointer is loaded [**DevBW, DevCL**] Note that VFE_STATE_EX is reserved
2	31:5	**Pointer to VFE_STATE.** Specifies the 32-byte aligned address of the VFE_STATE. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4:0	Reserved : MBZ

10.7.2 MEDIA_OBJECT Command

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline. It supports loading of inline data as well as indirect data.

The MEDIA_OBJECT command can be used in the following three VFE modes: Generic mode, IS mode and VLD mode.

The MEDIA_OBJECT command cannot be used in the following VFE modes: AVC-IT, AVC-MC, and VC1-IT.

Dword	Bits	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_OBJECT Pipeline[28:27] = Media = 2h; Opcode[26:24] **=** 1h; Subopcode[23:16] = 0h
	15:0	**DWord Length** (Excludes DWords 0,1) **VLD Mode:** DWord Length = 4. There are 2 DW of inline data in this mode. **IS Mode:** DWord Length = N+2, where N is the number of DW of inline data (N>= 10). According to the inline format table shown in the following section, N is 10. However, hardware must be able to handle different size of N, as software may determine later to transfer additional driver/kernel information inline. **Generic Mode:** DWord Length = N+2, where N is in the range of [0,504]. The maximum is 504 DW (equivalent to 63 8-DW registers). When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than or equal to 63 (with both inline data length N and indirect data length rounded up to 8-DW aligned individually). If indirect data are fetched, the minimal inline data length is 0. If indirect data are not fetched, the minimal inline data is 1DW. **Note:** Regardless of the mode, inline data must be present in this command.
1	31:8	**Debug**: **Object ID.** This field is used to initialize the VFE debug counter controlled by the VFE state.
	7	Reserved. MBZ
	6:0	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. *In VLD mode, this field is ignored by hardware.* Format = U7
2	31:29	Reserved. MBZ
	28	**Retain Bit.** The hardware will keep the last 256-bit quantity of this indirect object in-use after the object transfer is complete. A subsequent Indirect object packet will use the retained 256bit quantity as the first piece of data. Format = Enable (1) /Disable (0) **[DevBW-A] Erratum: this field is reserved: MBZ**
	27:25	Reserved. MBZ

Dword	Bits	Description
	24	**Thread Synchronization.** This field when set indicates that the dispatch of the thread originated from this command is based on the “spawn root thread” message. *In VLD mode, this field must be programmed as 0, because the **Children Present** field in VFE_STATE must be 0 in this mode.* 0 = No thread synchronization 1 = Thread dispatch is synchronized by the “spawn root thread” message
	23:17	Reserved. MBZ
	16:0	**Indirect Data Length.** This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. This field must have the same alignment as the Indirect Object Data Start Address. **VLD Mode:** It is the length in bytes of the bitstream data for the current slice. It includes the first byte of the first macroblock and the last non-zero byte of the last macroblock in the slice. Specifically, the zero-padding bytes (if present) and the next start-code are excluded. Hardware ignores the contents after the last non-zero byte. This field is sized to support MPEG-2 MP@HL bitstream. According to Table 8-6 of *ISO/IEC 13818-2,* the maximum number of bits per macroblock for 4:2:0 is 4608. So the maximum slice size for MP@HL (e.g. 1080i) is 4608 * 120 / 8 = 69120 bytes (0x10E00), which requires 17 bits. **IS Mode:** It must be DWord aligned. **Generic Mode:** It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than or equal to 63 (with both inline data length and indirect data length rounded up to 8-DW aligned). [**DevBW-A**] Erratum: In Generic Mode, the length alignment restrict is relaxed to be DWord alignment.. Format = U17 in bytes
3	31:0	**Indirect Data Start Address.** This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. Alignment of this address depends on the mode of operation. **VLD Mode:** It is the byte aligned address for the VLD bitstream data. **IS Mode:** It is the DWord aligned address for the first IDCT coefficients. **Generic Mode:** It is the DWord aligned address of the indirect data. Range = [0 - 512MB] (Bits 31:29 MBZ)
4..N	31:0	**Inline Data** **IS and VLD Modes:** Hardware interprets this data in the specified format. **Generic Mode:** The format of this data is specified by software. Hardware does not interpret this data; it merely passes it to the kernel for processing. The total size for the inline data and indirect data must not exceed the URB allocation size.

10.7.2.1 Inline and Indirect Data Format in Generic Mode

In Generic mode, inline data must be present. All inline data will be delivered to the thread's payload starting and ending on the 8-DW aligned register boundary. Inline data starts on dword 4 of the MEDIA_OBJECT command. If the dword length field of the MEDIA_OBJECT command is N+2, the size of the inline data will N. VFE always zero-pads inline data into 8-DW before delivering to URB. If N is multiple of 8-DW, the inline data corresponds to exactly N/8 GRF registers. If N is not multiple of 8-DW, there will be (N/8 + 1) registers written for the inline data with the last register containing the last a few dwords of inline data with remaining dwords zeroed out by VFE.

Indirect data, if present, will be written into GRF registers immediately following the inline data in the thread's payload. Alignment and padding for indirect data are the same as that for inline data. In short, indirect data are also starting and ending on 8-DW aligned register boundary. If indirect data length is not multiple of 8-DW, VFE hardware will zero pad the last GRF register.

10.7.2.2 Inline and Indirect Data Format in IS Mode

Each MEDIA_OBJECT command in "IS mode" corresponds to the processing of one macroblock. Macroblock parameters are passed in as inline data and the non-zero DCT coefficient data for the macroblock is passed in as indirect data.

Table 10-9 depicts the inline data format in IS mode. All fields in inline data are forwarded to the thread as thread payload. Alignment and padding is identical to that described for Generic mode. Some fields are merely forwarded. Some fields are also used by VFE as indicated in the following table by a mark of [Used by VFE]. As shown, inline data starts at dword 4 of MEDIA_OBJECT command. There are 10 dwords total.

Table 10-9. Inline data in IS mode

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>4+0</td><td>31:28</td><td>Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the ISO/IEC 13818-2 (see also §7.6.4).
<table>
<tr><th>Bit</th><th>MVector [r]</th><th>MVector [s]</th><th>MotionVerticalFieldSelect Index</th></tr>
<tr><td>28</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>29</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>30</td><td>1</td><td>0</td><td>2</td></tr>
<tr><td>31</td><td>1</td><td>1</td><td>3</td></tr>
</table>
Format = MC_MotionVerticalFieldSelect.
0 = The prediction is taken from the top reference field.
1 = The prediction is taken from the bottom reference field.</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td>27</td><td>Second Field. This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the ISO/IEC 13818-2.

When the picture type is not P or the prediction type is not field, this value should be 0.
Format = MC_SecondPField
0 = This is not the second field.
1 = This is the second field.</td></tr>
<tr><td></td><td>26</td><td>Reserved. (HWMC mode)</td></tr>
<tr><td></td><td>25:24</td><td>Motion Type. When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See ISO/IEC 13818-2 §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type.

Format = MC_MotionType

<table>
<tr><th>Value</th><th>Destination = Frame

Picture_Structure = 11</th><th>Destination = Field

Picture_Structure != 11</th></tr>
<tr><td>‘00’</td><td>Reserved</td><td>Reserved</td></tr>
<tr><td>‘01’</td><td>Field</td><td>Field</td></tr>
<tr><td>‘10’</td><td>Frame</td><td>16x8</td></tr>
<tr><td>‘11’</td><td>Dual-Prime</td><td>Dual-Prime</td></tr>
</table></td></tr>
<tr><td></td><td>23:22</td><td>Reserved. (Scan method)</td></tr>
<tr><td></td><td>21</td><td>DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See ISO/IEC 13818-2 §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present).
0 = MC_FRAME_DCT (Macroblock is frame DCT coded).
1 = MC_FIELD_DCT (Macroblock is field DCT coded).</td></tr>
<tr><td></td><td>20</td><td>Overlap Transform (H261 Loop Filter). This field, when set, indicates that overlap smoothing filter is performed after motion compensation and before in-loop deblocking.</td></tr>
<tr><td></td><td>19</td><td>4MV Mode: (H263)

This field indicates if the current macroblock is coded with 4 motion vectors, one for each 8x8 block.</td></tr>
<tr><td></td><td>18</td><td>Macroblock Motion Backward. This field specifies if the backward motion vector is active. See ISO/IEC 13818-2 Tables B-2 through B-4.
0 = No backward motion vector.
1 = Use backward motion vector(s).</td></tr>
<tr><td></td><td>17</td><td>Macroblock Motion Forward. This field specifies if the forward motion vector is active. See ISO/IEC 13818-2 Tables B-2 through B-4.
0 = No forward motion vector.
1 = Use forward motion vector(s).</td></tr>
</table>

DWord	Bit	Description
	16	**Macroblock Intra Type.** This field specifies if the current macroblock is intra-coded. When set, Coded Block Pattern is ignored and no prediction is performed (i.e., no motion vectors are used). See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:0	Reserved. (MB address)
4+1	31:24	Reserved. (Skip Macroblocks)
	23:0	Reserved. (Offset into error data)
4+2	31:24	**Subblock Coding for Block Y1**
	23:16	Subblock Coding for Block Y0. This field specifies the subblock partition and subblock coding pattern for the block. The definition of the 8 bits of this field is listed below. Detailed coding can be found in Table 10-10. Bits [7:6]: reserved Bits [5:2]: Subblock present Bits [1:0]: Subblock partitioning
	15:12	Reserved.
	11:6	**Coded Block Pattern.** This field specifies whether blocks are present or not. Format = 6-bit mask. Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb4 Bit 6: Cr5 [Used by VFE]
	5:0	Reserved. (Quantization Scale Code)
4+3	31:24	**Subblock Coding for Block Cr5**
	23:16	**Subblock Coding for Block Cb4**
	15:8	**Subblock Coding for Block Y3**
	7:0	**Subblock Coding for Block Y2**
4+4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
4+5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**

DWord	Bit	Description
4+6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
4+7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
4+8	31:30	Reserved.
	29	Reserved
	28	Reserved
	27:20	Reserved.
	19:18	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	Reserved. (8-bit Intra)
	14:13	Reserved. (Intra DC Precision)
	12:0	Reserved.
4+9	31:27	Reserved.
	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120). Format = U7 in macroblock units. Range = [0, 120]
	19:11	Reserved: MBZ
	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks. Format = U7 in macroblock units. Range = [0, 127]
	3:0	Reserved.

The control parameters for inverse-scan are carried in the inline data packet. In particular, the Coded Block Pattern field in DW6 is used to determine how many blocks are coded and therefore how many blocks will be output from the inverse-scan.

Besides that dword 6 (containing Coded Block Pattern field) is used by VFE hardware as control parameter for inverse-scan, the rest of the inline data are determined between the host software and the kernel software. Therefore, the exact format and size of the inline data may differ, as long as the Coded Block Pattern field in dword 6 remains the same.

Table 10-9 shows a 'recommended' inline data format. Support for other video coding standard may take a similar format. For each block, the subblock coding occupies one byte with only the lower 6 bits utilized according to Table 10-10. For MPEG-2 IDCT support, as there is no block subdivision, these fields may be reserved, or used to carry other information.

Table 10-10. Subblock coding (bits [7:6] are reserved).

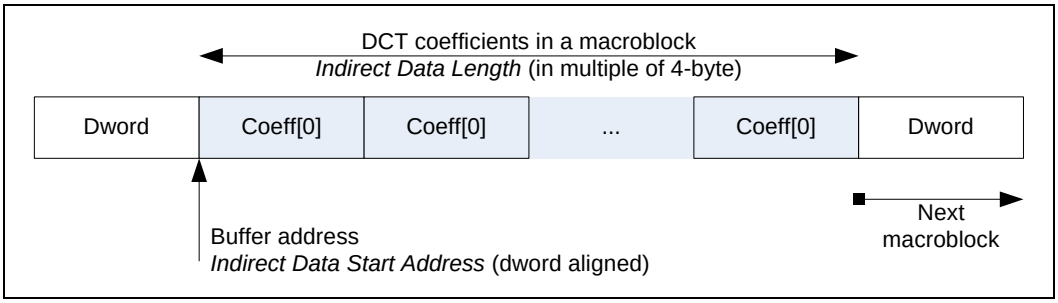
Subblock Partitioning (Bits [1:0])		Subblock Present (0 means not present, 1 means present)			
Code	**Meaning**	**Bit 2**	**Bit 3**	**Bit 4**	**Bit 5**
00	Single 8x8 block (sb0)	Sb0	Don't care	Don't care	Don't care
01	Two 8x4 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
10	Two 4x8 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
11	Four 4x4 subblocks (sb0-3)	Sb0	Sb1	Sb2	Sb3

The block data output from the inverse scan will follow immediately after the inline data in the thread's payload, again, aligning to GRF register. Block data output by nature is 8-DW aligned. The actual size depends on the coded block pattern. As each block contains 8x8 16-bit DCT coefficients, if the total number of coded block is M, the block data will take 8 * 8 * 2 * M / 32 = 4 * M GRE registers.

As VFE performs inverse-scan on the indirect data, the indirect data must follow the exact format described in Figure 10-9 and Table 10-11.

The indirect data start address in MEDIA_OBJECT specifies the doubleword aligned address of the first non-zero DCT coefficient of the first block of the macroblock. Only the non-zero DCT coefficients are present in the data buffer and they are packed in the block sequence of Y0, Y1, Y2, Y3, Cb4 and Cr5, as shown in Figure 10-9. The indirect data length in MEDIA_OBJECT includes all the non-zero coefficients for the macroblock. It must be doubleword aligned.

Figure 10-9. Structure of the IDCT Compressed Data Buffer



Each non-zero coefficient in the indirect data buffer is contained in a doubleword-size data structure containing the coefficient index, end of block (EOB) flag and the fixed-point coefficient value in 2's complement form. As shown in Table 10-11, *index* is the row major 'raster' index of the coefficient within an 8x8 block. DCT coefficient is a 16-bit value in 2's complement, which is clamped to a 12-bit signed value by the host. Effectively, bit 27 is the sign bit. However, as the kernel software consumes these data as 16-bit quantities any way, VFE simply forwards these exact 16-bit DCT coefficients to the thread's payload.

Table 10-11. Structure of a DCT coefficient unit

DWord	Bit	Description
0	31:16	**DCT Coefficient Value.** This field contains the value of the non-zero DCT coefficient in 2's complement.
	15:7	Reserved: MBZ
	6:1	**Index.** This field specifies the raster-scan address (raw address) of the DCT coefficient within the 8x8 block. For example, coefficient at location (row, column) = (0, 0) has an index of 0; that at (2, 3) has an index of 2*8 + 3 = 19. Format = U6 Range = [0, 63]
	0	**EOB (End of Block).** This field indicates whether the DCT coefficient is the last one of the current block.

10.7.2.3 Inline and Indirect Data Format in VLD Mode

A MEDIA_OBJECT command in “VLD mode” is used to process a slice using the VFE hardware. Slice header parameters are passed in as inline data and the bitstream data for the slice is passed in as indirect data. Of the inline data, slice_horizontal_position and slice_vertical_position determines the location within the destination picture of the first macroblock in the slice.

DWord	Bits	Description
4	31	Reserved. MBZ
	30:24	**Slice Horizontal Position.** This 7-bit field indicates the horizontal position (in macroblock units) of the first macroblock in the slice. Format = U7 in macroblocks
	23	Reserved. MBZ
	22:16	**Slice Vertical Position.** This 7-bit field indicates the vertical position (in macroblock units) of the first macroblock in the slice. Format = U7 in macroblocks
	15	Reserved. MBZ
	14:8	**Macroblock Count.** This 7-bit field indicates the number of macroblocks in the slice, including skipped macroblocks.
	7:3	Reserved. MBZ.
	2:0	**First Macroblock Bit Offset.** This field provides the bit offset of the first macroblock in the first byte of the input bitstream. Format = U3
5	31:29	Reserved. MBZ.
	28:24	**Quantizer Scale Code.** This field sets the quantizer scale code of the inverse quantizer. It remains in effect until changed by a decoded quantizer scale code in a macroblock. This field is decoded from the slice header by host software. Format = U5 (0 is Reserved)
	23:0	Reserved. MBZ.

The indirect data start address in MEDIA_OBJECT specifies the starting Graphics Memory address of the bitstream data that follows the slice header. It provides the byte address for the first macroblock of the slice. Together with the First Macroblock Bit Offset field in the inline data, it provides the bit location of the macroblock within the compressed bitstream.

The indirect data length in MEDIA_OBJECT provides the length in bytes of the bitstream data for this slice. It includes the first byte of the first macroblock and the last **non-zero** byte of the last macroblock in the slice. Specifically, the zero-padding bytes (if present) and the next start-code are excluded. Hardware ignores the contents after the last non-zero byte. Figure 10-10 illustrates these parameters for a slice data.

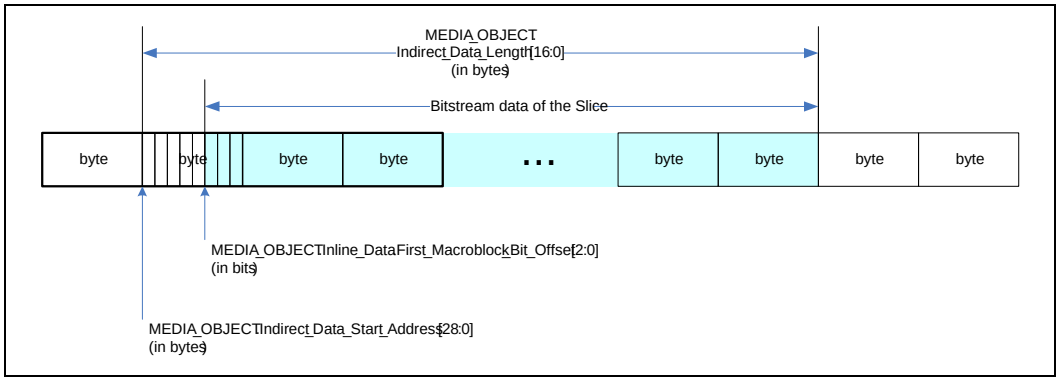


Figure 10-10. Indirect data buffer for a slice

10.8 Media Messages

All message formats are given in terms of dwords (32 bits) using the following conventions which are detailed in GEN4 Subsystem Chapter.

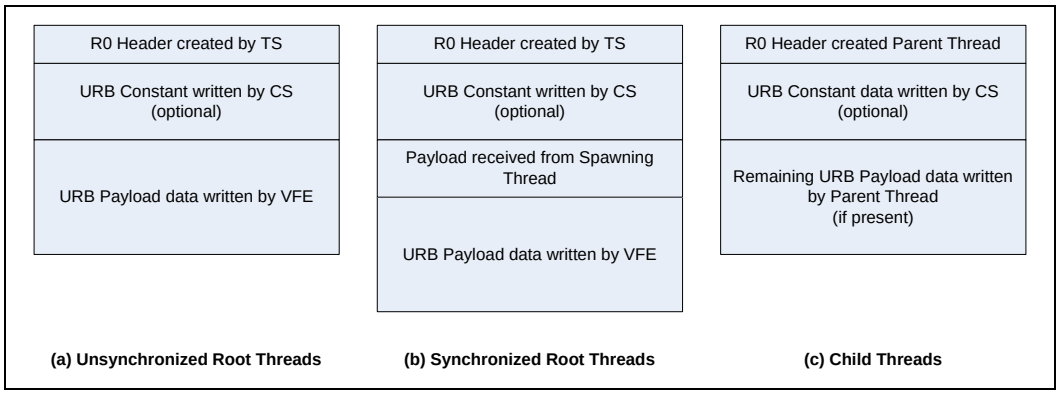
Dispatch Messages: **R**p.d

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

10.8.1 Thread Payload Messages

The root thread's register contents differ from that of child threads, as shown in Figure 10-11. The register contents for a synchronized root thread (also referred to as 'spawned root thread') and an unsynchronized one are also different. Whether the URB Constant data field is present or not is determined by the interface descriptor of a given thread. This applies to both root and child threads. When URB Constant data field is present for a synchronized root thread, URB constant data field is before the data field received from the spawning thread, which is also before the URB payload data.

Figure 10-11. Thread payload message formats for root and child threads



10.8.1.1 Generic Mode Root Thread

The following table shows the R0 register contents for a Generic mode root thread, which is generated by TS. The remaining payloads are application dependent.

Table 10-12. R0 header of a generic mode root thread

DWord	Bit	Description
R0.7	31	**Debug** : **Snapshot Flag**. This field is used by the Thread Dispatcher to set the snapshot flag upon a snapshot condition.
	27:24	**Debug** : Reserved
	23:0	**Debug** : Reserved for Parent Thread Count. Root threads should have zero in this field.
R0.6	31:24	**Debug** : Reserved for software debug. This field is reserved for the system debug routine, for example, to assemble the debug scratch memory offset for the thread. Fixed function hardware and application routine must not use this field.
	23:0	**Debug** : **Thread Count.** This field is generated by VFE based on a debug counter that is controlled by host software.
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers from. Format = GeneralStateOffset[31:4]
	3:0	Reserved : MBZ
R0.1	31:28	Reserved : MBZ

DWord	Bit	Description
	27:26	Reserved : MBZ
	25	Reserved. MBZ
	24:16	Reserved : MBZ
	15:12	Reserved : MBZ
	11:9	Reserved. MBZ
	8:0	Reserved : MBZ
R0.0	31:24	Reserved : MBZ
	23:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

10.8.1.2 IS-Mode Root Thread

The following table shows the root thread payload messages when VFE is in IS mode and URB push constant is not enabled.

DWord	Bit	Description
R0.7	31	**Debug** : **Snapshot Flag**. This field is used by the Thread Dispatcher to set the snapshot flag upon a snapshot condition.
	27:24	**Debug** : Reserved
	23:0	**Debug** : MBZ.
R0.6	31:24	**Debug** : Reserved for software debug. This field is reserved for the system debug routine, for example, to assemble the debug scratch memory offset for the thread. Fixed function hardware and application routine must not use this field.
	23:0	**Debug** : **Thread Count.** This field is generated by VFE based on a debug counter that is controlled by host software.
R0.5	31:10	**NOT USED** (was Scratch Space Pointer).
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion. Note: Nothing to free up in this case.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ
R0.3	31:5	**NOT USED** (was Sampler State Pointer).
	4	Reserved : MBZ

DWord	Bit	Description
	3:0	**NOT USED** (was Per Thread Scratch Space)
R0.2	31:5	**Interface Descriptor Pointer.** Specifies the 32-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers from. Format = GeneralStateOffset[31:5]
	4:0	Reserved : MBZ
R0.1	31:0	Reserved : MBZ
R0.0	31:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children. This may be used if child threads and/or synchronized root threads are present in IS mode.
R1.7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
R1.6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
R1.5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**
R1.4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
R1.3	31:24	**Subblock Coding for Block Cr5**
	23:16	**Subblock Coding for Block Cb4**
	15:8	**Subblock Coding for Block Y3**
	7:0	**Subblock Coding for Block Y2**
R1.2	31:24	**Subblock Coding for Block Y1**
	23:16	**Subblock Coding for Block Y0.** This field specifies the subblock partition and subblock coding pattern for the block. The definition of the 8 bits of this field is listed below. Detailed coding can be found in Table 10-10. Bits [7:6]: reserved Bits [5:2]: Subblock present Bits [1:0]: Subblock partitioning
	15:12	Reserved.

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td>11:6</td><td>Coded Block Pattern. This field specifies whether blocks are present or not.
Format = 6-bit mask.
Bit 11: Y0
Bit 10: Y1
Bit 9: Y2
Bit 8: Y3
Bit 7: Cb5
Bit 6: Cr5</td></tr>
<tr><td></td><td>5:0</td><td>Reserved.</td></tr>
<tr><td>R1.1</td><td>31:24</td><td>Reserved. (Skip Macroblocks)</td></tr>
<tr><td></td><td>23:0</td><td>Reserved. (Offset into error data)</td></tr>
<tr><td>R1.0</td><td>31:28</td><td>Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the ISO/IEC 13818-2 (see also §7.6.4).

<table>
<tr><th>Bit</th><th>MVector [r]</th><th>MVector [s]</th><th>MotionVerticalFieldSelect Index</th></tr>
<tr><td>28</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>29</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>30</td><td>1</td><td>0</td><td>2</td></tr>
<tr><td>31</td><td>1</td><td>1</td><td>3</td></tr>
</table>

Format = MC_MotionVerticalFieldSelect.
0 = The prediction is taken from the top reference field.
1 = The prediction is taken from the bottom reference field.</td></tr>
<tr><td></td><td>27</td><td>Second Field. This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the ISO/IEC 13818-2.
When the picture type is not P or the prediction type is not field, this bit is set to 0.
Format = MC_SecondPField
0 = This is not the second field.
1 = This is the second field.</td></tr>
<tr><td></td><td>26</td><td>Reserved. (HWMC mode)</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td>25:24</td><td>Motion Type. When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See ISO/IEC 13818-2 §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type.
Format = MC_MotionType
<table><tr><th>Value</th><th>Destination = Frame

Picture_Structure = 11</th><th>Destination = Field

Picture_Structure != 11</th></tr><tr><td>‘00’</td><td>Reserved</td><td>Reserved</td></tr><tr><td>‘01’</td><td>Field</td><td>Field</td></tr><tr><td>‘10’</td><td>Frame</td><td>16x8</td></tr><tr><td>‘11’</td><td>Dual-Prime</td><td>Dual-Prime</td></tr></table></td></tr>
<tr><td></td><td>23:22</td><td>Reserved. (Scan method)</td></tr>
<tr><td></td><td>21</td><td>DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See ISO/IEC 13818-2 §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present).
0 = MC_FRAME_DCT (Macroblock is frame DCT coded).
1 = MC_FIELD_DCT (Macroblock is field DCT coded).</td></tr>
<tr><td></td><td>20</td><td>Reserved. (H261 Loop Filter)</td></tr>
<tr><td></td><td>19</td><td>Reserved. (H263)</td></tr>
<tr><td></td><td>18</td><td>Macroblock Motion Backward. This field specifies if the backward motion vector is active. See ISO/IEC 13818-2 Tables B-2 through B-4.
0 = No backward motion vector.
1 = Use backward motion vector(s).</td></tr>
<tr><td></td><td>17</td><td>Macroblock Motion Forward. This field specifies if the forward motion vector is active. See ISO/IEC 13818-2 Tables B-2 through B-4.
0 = No forward motion vector.
1 = Use forward motion vector(s).</td></tr>
<tr><td></td><td>16</td><td>Macroblock Intra Type. This field specifies if the current macroblock is intra-coded. When set, Coded Block Pattern is ignored and no prediction is performed (i.e., no motion vectors are used). See ISO/IEC 13818-2 Tables B-2 through B-4.
0 = Non-intra macroblock.
1 = Intra macroblock.</td></tr>
<tr><td></td><td>15:0</td><td>Reserved.</td></tr>
<tr><td>R2.7</td><td>31:0</td><td>Reserved.</td></tr>
<tr><td>R2.6</td><td>31:0</td><td>Reserved.</td></tr>
<tr><td>R2.5</td><td>31:0</td><td>Reserved.</td></tr>
<tr><td>R2.4</td><td>31:0</td><td>Reserved.</td></tr>
<tr><td>R2.3</td><td>31:0</td><td>Reserved.</td></tr>
<tr><td>R2.2</td><td>31:0</td><td>Reserved.</td></tr>
</table>

DWord	Bit	Description
R2.1	31:27	Reserved.
	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120). Format = U7 in macroblock units. Range = [0, 120]
	19:11	Reserved: MBZ
	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks. Format = U7 in macroblock units. Range = [0, 127]
	3:0	Reserved.
R2.0	31:30	Reserved.
	29	Reserved
	28	Reserved
	27:20	Reserved.
	19:18	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	Reserved. (8-bit Intra)
	14:13	Reserved. (Intra DC Precision)
	12:0	Reserved.

DWord	Bit	Description
None (0 blocks coded) or R3-R[2+4x] where x = number of coded blocks		**DCT Coefficients.** These are the DCT values of the coefficients for the macroblock. Only coded blocks have coefficients present in the array. Beginning in R3, the order of the coefficients for the coded blocks is Y0, Y1, Y2, Y3, Cb4, and Cr5. For each coded block, the 8x8 DCT coefficients, with 1 word each coefficient, are organized in row-major order, occupying four GRF registers. This is shown in Table 10-13, where the index-pair for a DCT coefficient is (Column_Index, Row_Index).

10.8.1.3 VLD-Mode Root Thread

The following table shows the root thread payload messages when VFE is in VLD mode and URB push constant is not enabled. When URB push constant is enabled, it will start at R1. Subsequently, macroblock data starting with motion vectors will be put in GRF registers after the URB push constants.

DWord	Bit	Description
R0.7	31	**Debug** : **Snapshot Flag**. This field is used by the Thread Dispatcher to set the snapshot flag upon a snapshot condition.
	27:24	**Debug** : Reserved
	23:0	**Debug** : MBZ.
R0.6	31:24	**Debug** : Reserved for software debug. This field is reserved for the system debug routine, for example, to assemble the debug scratch memory offset for the thread. Fixed function hardware and application routine must not use this field.
	23:0	**Debug** : **Thread Count.** This field is generated by VFE based on a debug counter that is controlled by host software.
R0.5	31:10	**NOT USED (was Scratch Space Pointer).**
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion. Note: Nothing to free up in this case.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ

DWord	Bit	Description
R0.3	31:5	**NOT USED (was Sampler State Pointer).**
	4	Reserved : MBZ
	3:0	**NOT USED (was Per Thread Scratch Space)**
R0.2	31:5	**Interface Descriptor Pointer.** Specifies the 32-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers from. Format = GeneralStateOffset[31:5]
	4:0	Reserved : MBZ
R0.1	31:0	Reserved : MBZ
R0.0	31:16	Reserved : MBZ
	15:0	**NOT USED (was URB Handle)**
R1.7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
R1.6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
R1.5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**
R1.4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
R1.3	31:27	Reserved.
	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120). Format = U7 in macroblock units. Range = [0, 120]
	19:11	Reserved: MBZ
	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks. Format = U7 in macroblock units. Range = [0, 127]
	3:0	Reserved.
R1.2	31:30	Reserved.
	29	Reserved. (Interpolation Rounder Control)
	28	Reserved. (Bidirectional Averaging Control)

DWord	Bit	Description
	27:20	Reserved.
	19:18	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	Reserved. (8-bit Intra)
	14:13	**Intra DC Precision.** See ISO/IEC 13818-2 §6.3.10 for details.
	12	**Disable Mismatch.** This bit is used to disable the mismatch control performed after the inverse quantization operation, as described in ISO/IEC 13818-2 §7.4.4
	11:6	**Coded Block Pattern.** This field specifies whether blocks are present or not. Format = 6-bit mask. Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb5 Bit 6: Cr5
	5	**Quantizer Scale Type**: This field specifies the quantizer scaling type. Format = MPEG_Q_SCALE_TYPE 0 = MPEG_QSCALE_LINEAR 1 = MPEG_QSCALE_NONLINEAR
	4:0	**Quantization Scale Code.** Combined with the quantization scale type, this value selects the quantizer scale table according to ISO/IEC 13818-2 Table 7-6
R1.1	31:24	Reserved. (Skip Macroblocks)
	23:0	Reserved. (Offset into error data)

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>R1.0</td><td>31:28</td><td>Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the ISO/IEC 13818-2 (see also §7.6.4).

<table>
<tr><th>Bit</th><th>MVector [r]</th><th>MVector [s]</th><th>MotionVerticalFieldSelect Index</th></tr>
<tr><td>28</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>29</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>30</td><td>1</td><td>0</td><td>2</td></tr>
<tr><td>31</td><td>1</td><td>1</td><td>3</td></tr>
</table>

Format = MC_MotionVerticalFieldSelect.
0 = The prediction is taken from the top reference field.
1 = The prediction is taken from the bottom reference field.</td></tr>
<tr><td></td><td>27</td><td>Second Field. This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the ISO/IEC 13818-2.

When the picture type is not P or the prediction type is not field, this bit is set to 0.
Format = MC_SecondPField
0 = This is not the second field.
1 = This is the second field.</td></tr>
<tr><td></td><td>26</td><td>Reserved. (HWMC mode)</td></tr>
<tr><td></td><td>25:24</td><td>Motion Type. When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See ISO/IEC 13818-2 §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type.

Format = MC_MotionType

<table>
<tr><th>Value</th><th>Destination = Frame Picture_Structure = 11</th><th>Destination = Field Picture_Structure != 11</th></tr>
<tr><td>‘00’</td><td>Reserved</td><td>Reserved</td></tr>
<tr><td>‘01’</td><td>Field</td><td>Field</td></tr>
<tr><td>‘10’</td><td>Frame</td><td>16x8</td></tr>
<tr><td>‘11’</td><td>Dual-Prime</td><td>Dual-Prime</td></tr>
</table></td></tr>
<tr><td></td><td>23:22</td><td>Reserved. (Scan method)</td></tr>
<tr><td></td><td>21</td><td>DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See ISO/IEC 13818-2 §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present).
0 = MC_FRAME_DCT (Macroblock is frame DCT coded).
1 = MC_FIELD_DCT (Macroblock is field DCT coded).</td></tr>
<tr><td></td><td>20</td><td>Reserved. (H261 Loop Filter)</td></tr>
<tr><td></td><td>19</td><td>Reserved. (H263)</td></tr>
</table>

DWord	Bit	Description
	18	**Macroblock Motion Backward.** This field specifies if the backward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No backward motion vector. 1 = Use backward motion vector(s).
	17	**Macroblock Motion Forward.** This field specifies if the forward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No forward motion vector. 1 = Use forward motion vector(s).
	16	**Macroblock Intra Type.** This field specifies if the current macroblock is intra-coded. When set, Coded Block Pattern is ignored and no prediction is performed (i.e., no motion vectors are used). See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:0	Reserved.
None (0 block coded) or R2-R[1+4x] where x = number of coded blocks		**DCT Coefficients.** These are the DCT values of the coefficients for the macroblock. Only coded blocks have coefficients present in the array. Beginning in R2, the order of the coefficients for the coded blocks is Y0, Y1, Y2, Y3, Cb4, and Cr5. For each coded block, the 8x8 DCT coefficients, with 1 word each coefficient, are organized in row-major order, occupying four GRF registers. This is shown in Table 10-13, where the index-pair for a DCT coefficient is (Column_Index, Row_Index).

Table 10-13. Format of a block of DCT coefficients in GRF registers

Reg. / Words	W15	W14	W13	W12	W11	W10	W9	W8	W7	W6	W5	W4	W3	W2	W1	W0
R[n]	(7,1)	(6,1)	(5,1)	(4,1)	(3,1)	(2,1)	(1,1)	(0,1)	(7,0)	(6,0)	...	...	...	...	(1,0)	(0,0)
R[n+1]	(7,3)	...	...	...	...	...	...	(0,3)	(7,2)	(6,2)	...	...	...	...	(1,2)	(0,2)
R[n+2]	(7,5)	...	...	...	...	...	...	(0,5)	(7,4)	(6,4)	...	...	...	...	(1,4)	(0,4)
R[n+3]	(7,7)	(7,7)	(5,7)	(4,7)	(3,7)	(2,7)	(1,7)	(0,7)	(7,6)	(6,6)	...	...	...	...	(1,6)	(0,6)

* W# (# from 0 to 15) represents WORD location # within an 8-DW register.

10.8.1.4 Child Thread

The tread initiation for the child thread is determined by the data stored in the URB by the parent that spawns it. No hardware-defined header is generated. Besides the debug fields, all other fields are software defined according to application specific needs. However, software should follow the header field definition similar to that for a root thread, when the same fields are used, to be consistent and to reduce message header assemble overhead.

As the parent thread is responsible of generating the debug fields, it should follow the same general principle used by fixed function hardware. The Parent Thread Count field should be the Thread Count field of the parent thread itself (e.g. copying R0.6[23:0] to R0.7[23:0]). The Thread Count field should have a unique value for each child thread and the unique value should not be dependent on the execution order. This is mostly important for the cases when the child thread generation order may vary depending on the thread completion order. For example, when generating child threads for macroblock-based processing, the Thread Count field for a child thread should be deterministic for a macroblock position.

The following table shows the R0 register contents for a child thread, which is generated by its parent thread. The remaining payloads are application dependent.

DWord	Bit	Description
R0.7	31	**Debug**: **Snapshot Flag**. This field is used by the Thread Dispatcher to set the snapshot flag upon a snapshot condition.
	27:24	**Debug**: Reserved
	23:0	**Debug**: **Parent Thread Count.** This field is the thread count of the parent thread that can be used to uniquely identify the parent thread that generates this child thread.
R0.6	31:24	**Debug**: Reserved for software debug. This field is reserved for the system debug routine, for example, to assemble the debug scratch memory offset for the thread. Fixed function hardware and application routine must not use this field.
	23:0	**Debug**: **Thread Count.** This field is generated by the parent thread to uniquely identify the child thread.
R0.5-R0.0	31:0	**Software defined**

10.8.2 Thread Spawn Message

The thread spawn message is issued to the TS unit by a thread running on an EU. This message contains only one 8-DW register. The thread spawn message may be used to

- Spawn a child thread
- Spawn a root thread (start dispatching a synchronized root thread)
- Dereference URB handle
- Indicate a thread termination, dereference other TS managed resource and may or may not dereference URB handle

In order to end a root thread, the end of thread message must be targeted at the thread spawner. In this case, the root thread sends a message with a “dereference resource” in the Opcode field. The thread spawner does *not* snoop the messages sideband to determine when a root thread has ended. Thread Spawner does not track when a child thread terminates, to be consistent a child thread should also terminate with a “dereference resource” message to the Thread Spawner. Software must set the Requester Type (root or child thread) field correctly.

As TS dispatches one synchronized root thread upon receiving a ‘spawn root thread’ message (from a synchronization thread). The synchronizing thread must send the number of ‘spawn root thread’ message exactly the same as the subsequent ‘synchronized root thread’. No more, no less. Otherwise, hardware behavior is undefined.

URB Handle Offset field in this message (in M0.4) has 10 bits, allowing addressing of a large URB space. However, when a parent thread writes into the URB, it subjects to the maximum URB offset limitation of the URB write message, which is only 6 bits (see Unified Return Buffer Chapter for details). In this case, the parent thread may have to modify the URB Return Handle 0 field of the URB write message in order to subdivide the large URB space that the thread manages.

10.8.2.1 Message Descriptor

The following table shows the lower 16 bits of the message descriptor within the SEND instruction for a thread spawn message.

Bit	Description
19	This bit is not part of the shared function specific message descriptor**.**
18:5	Reserved: MBZ**.** Bits 18:16 are not part of the shared function specific message descriptor.
4	**Resource Select.** This field specifies the resource associated with the action taken by the Opcode. If Opcode is “Spawn thread”, this field selects whether it is a child thread or a root thread. 0 = spawn a *child* thread 1 = spawn a *root* thread If Opcode == “Dereference Resource”, this field indicates whether the URB handle is to be dereferenced. The URB handle can only be dereferenced once. 0 = The URB handle is dereferenced 1 = The URB handle is NOT dereferenced
3:2	Reserved: MBZ
1	**Requester Type.** This field indicates whether the requesting thread is a root thread or a child thread. If it is a root thread, when Opcode is 0, FF managed resources will be dereferenced. If it is a child thread and Opcode is 0, no resource will be dereferenced – basically no action is required by the TS. 0 = Root thread 1 = Child thread
0	**Opcode.** Indicates the operation performed by the message. A root thread must terminate with a message to TS (Opcode == 0 and EOT == 1). A child thread *should* also terminate with such a message. A thread cannot terminate with an Opcode of “spawn thread”. 0 = dereference resource (also used for end of thread) 1 = spawn thread

10.8.2.2 Message Payload

DWord	Bit	Description
M0.7	31:0	**Debug**: Identical to DW7 of R0 of the requesting thread. Exception: If **Opcode** (and **Requester Type**) is "spawn a child thread", this field must be identical to DW7 of R0 of the **child thread** to be spawned in order for TS to generate debug snapshot for the child thread.
M0.6	31:0	**Debug**: Identical to DW6 of R0 of the requesting thread. Exception: If **Opcode** (and **Requester Type**) is "spawn a child thread", this field must be identical to DW6 of R0 of the **child thread** to be spawned in order for TS to generate debug snapshot for the child thread.
M0.5	31:8	Ignored
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by a root thread upon thread completion. This field is valid only if the **Opcode** is "dereference resource", and is ignored by hardware otherwise.
M0.4	31:16	Ignored
	15:10	**Dispatch URB Length.** Indicates the number of 8-DW URB entries contained in the Dispatch URB Handle that will be dispatched. When spawning a child thread, the URB handle contains most of the child thread's payload including R0 header. When spawning a root thread, the URB handle contains the message passed from the requesting thread to the spawned "peer" root thread. The number of GRF registers that will be initialized at the start of the spawned child thread is the addition of this field and the number of URB constants if present. The number of GRF registers that will be initialized at the start of a spawned root thread is the addition of this field, the number of URB constants if present, and the URB handle received from VFE. This field is ignored if the **Opcode** is "dereference resource". Length of 0 can be used only while spawning child threads to indicate that there is no payload beyond the required R0 header. It is UNDEFINED to set this field to 0 when spawning a root thread. Format = U6 Range = [0,63] for child threads Range = [1,63] for root threads
	9:0	**URB Handle Offset.** Specifies the 8-DW URB entry offset into the URB handle that determines where the associated dispatch payload will be retrieved from when the spawned child or root thread is dispatched. This field is ignored if the **Opcode** is "dereference resource". Format = U10 Range = [0,1023]
M0.3	31:0	Ignored

DWord	Bit	Description
M0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *the child thread's* interface descriptor. This pointer is used by TS to fetch the interface descriptor for the child thread, and it is also passed to the child thread in its R0 header. This field is ignored if the **Opcode** is "dereference resource" or "spawn a root thread". Format = GeneralStateOffset[31:4]
	3:0	Ignored
M0.1	31:0	Ignored
M0.0	31:24	Ignored
	23:16	Reserved : MBZ
	15:0	**Dispatch URB Handle** If **Opcode** (and **Requester Type**) is "spawn a child thread": Specifies the URB handle for the child thread. If **Opcode** (and **Requester Type**) is "spawn a root thread": Specifies the URB handle containing message (e.g. requester's gateway information) from the requesting thread to the spawned root thread. If **Opcode** is "dereference resource": This field is required on end of thread messages if the **Children Present** bit is set, as the handle must be dereferenced, otherwise this field is ignored.

10.9 Media Applications with Specific Hardware Support

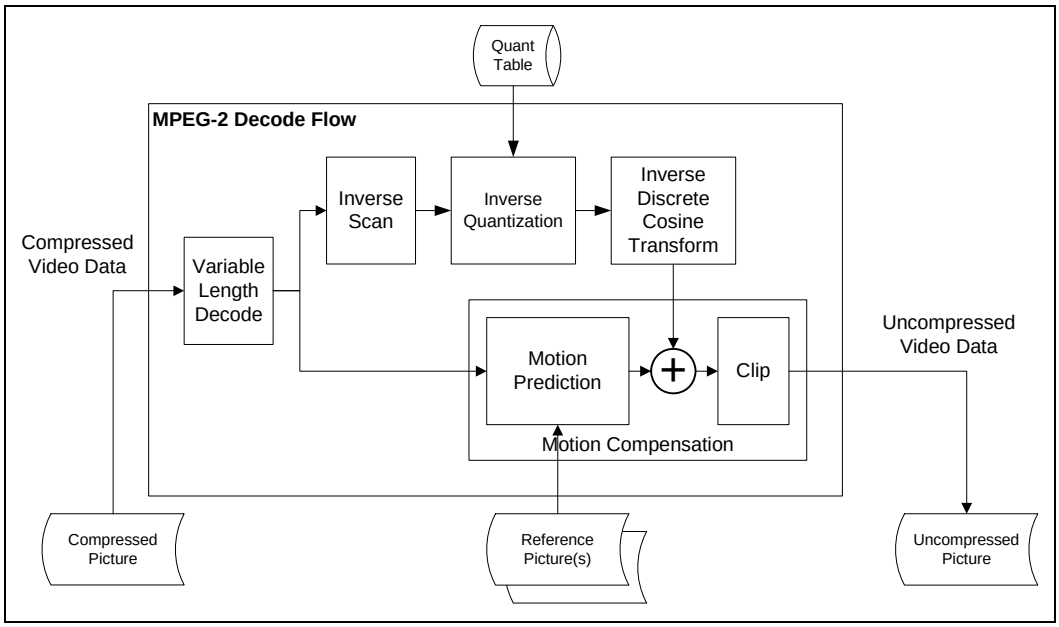
10.9.1 Full MPEG-2 Decode

10.9.1.1 Theory of Operation

In this section, we start with an introduction of MPEG-2 decode pipeline and the structure of MPEG-2 bitstream. We then discuss the host (CPU) and the graphics accelerator partition with reference to common API definitions. This leads to the VLD hardware acceleration and the VLD Command interface. Next, we discuss the partition between the VLD hardware (part of the VFE fixed function unit) and the kernel software running on GEN4 execution unit. This leads to the definition of the Post-VLD kernel descriptors. Another host-graphics-accelerator partition, HWMC, will be covered in the next Section. Yet another host-graphics-accelerator partition, HW-IDCT, can also be supported using the Inverse Scan mode of VFE. However, as it is a less interesting usage model, we will not cover it in this document. MPEG-2 is a video compression standard based on block coding, exploiting spatial and temporal data redundancy in natural video data. Temporal redundancy is reduced by inter-picture motion estimation; spatial redundancy is reduced by applied Discrete Cosine Transform (DCT). The DCT transformed motion prediction residue if inter-predicted (or the pixel data if intra-coded) is compressed using (run-length) Huffman Variable Length Coding (VLC). Decoding a compressed MPEG-2 video stream follows the reverse of this process. As shown in Figure 10-12, MPEG-2 video decoding includes the following steps:

- Variable Length Decode (VLD): A process that uses inverse Huffman table loop-up to convert a compressed bitstream into picture structure parameters, motion information and quantized DCT coefficients.
- Inverse Scan (IS): A process that converts run-length location based on a selected scan order of the quantized DCT coefficients into block coordinates.
- Inverse Quantization (IQ): A process that restores the DCT coefficients based on selected (inverse) quantization table.
- Inverse DCT (IDCT): An inverse transform to restore the motion prediction residue (for predicted macroblocks) or the pixel data (for non-predicated macroblocks).
- Motion Compensation (MC): A process that includes motion prediction, residual data addition and the final clipping. Motion prediction process uses the motion vectors to reconstruct predicted data from the reference pictures. If present, the predicted data is added to the residue data. The final clipping process converts the pixel data from 9-bit signed precision into 8-bit unsigned range [0, 255] before storing into memory or presenting for display.

Figure 10-12. MPEG-2 decode flow chart



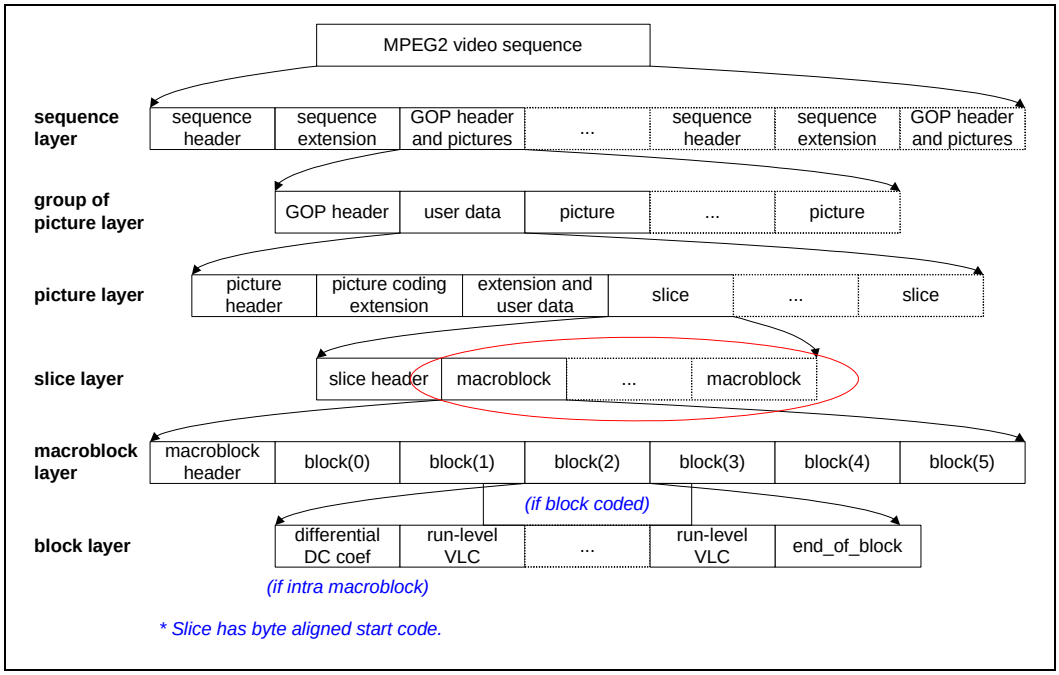
In order to reduce hardware complexity while maximizing the host acceleration, a proper host/hardware partition is desired. According to the MPEG-2 compressed bitstream syntax shown in Figure 10-13, the compressed bitstream is organized in a hierarchical structure. We observe that:

- All layers above slice layer start with unique byte aligned header start-code. The start codes can be easily found within the bitstream without decoding all the symbols in between.
- Majority of bits spent in the macroblock and block layer and there is no start code for these two layers. This indicates that most of bitstream parsing and VLD compute are spent in these two layers.

Based on these two observations, natural partition between host software and the graphics acceleration hardware is at the Slice layer if the hardware performs off-host

VLD operations. This partition provides the maximal host acceleration while also replying on host for high level bitstream syntax decoding to simplify the graphics accelerator's complexity. This host/accelerator partition is fully supported by common API interfaces.

Figure 10-13. MPEG-2 compressed bitstream syntax



The types of computation required for different MPEG-2 decoding stages are significantly different, calling for different hardware implementation. As VLD consists of a bit-wide variable length operation with table look up in the very inner loop, it is best suited for ASIC implementation – VLD hardware in the Video Front End (VFE) fixed function unit. In contrary, the rest of IQ, IDCT and MC stages are very well fitted for SIMD type programmable hardware – GEN4 Execution Unit. Inverse Scan fails in between. As it is a process of converting unstructured data into well-structured block data, performing it in VFE unit significantly reduces the burden of the kernels running on GEN4 EU. The mapping of the MPEG-2 decode process flow into GEN4 hardware is illustrated in Figure 10-14. This mapping can be summarized by the following points.

- VLD hardware performs Slice VLD and Inverse Scan. It takes Slice commands and compressed bitstream data from the Command Streamer. It outputs the decoded macroblock parameters and the quantized coefficient blocks into the URB as part of the post-VLD kernel descriptors. Upon the completion of a macroblock, VFE unit signals the TS unit that a new kernel is ready to be dispatched for execution.
- The per-macroblock post-VLD kernel descriptors in URB are then forward by the TS unit to the GEN4 Thread Dispatcher to be dispatched as GEN4 threads on GEN4 Execution Units.
- The post-VLD threads running on GEN4 EU performs IQ, IDCT and MC. The quantized IDCT block data is preloaded to the thread. The quantization tables and IDCT transform coefficients. mapped to the Data Cached are loaded from the Data

Port similar to the Constants used in 3D graphics. The reference blocks in the reference pictures mapped to the Texture Cache are loaded from the Data Port using Media Block Read message. The final resulting macroblock data are written via the Data Port using Media Block Write message to the uncompressed picture buffer that is mapped to the Render Cache.

- There are two ways of delivering quantization tables and IDCT transform coefficients: pulled in from data port, or pushed in via Constant URB buffer.
  - As shown in Figure 10-14, both quantization table and IDCT transform coefficient matrices may be mapped to the data cache and be loaded from data port by the thread. In this usage mode, data cache should be large enough to hold the data. Performance penalty for a thread is the round trip latency for cache hit case.
  - Alternatively and **preferably**, both quantization table and IDCT transform coefficient matrices may be pushed in the thread payload as Constant URB buffer. As both quantized IDCT block data and quantization tables are ready in the thread payload, thread program can proceed right away without any data due to data load. In addition, post-VLD kernels are relatively long kernels. They are not dispatch limited. Pushing these data via Constant URB buffer should not impact performance.

Figure 10-14. Functional mapping of MPEG-2 decode hardware acceleration with off-host VLD

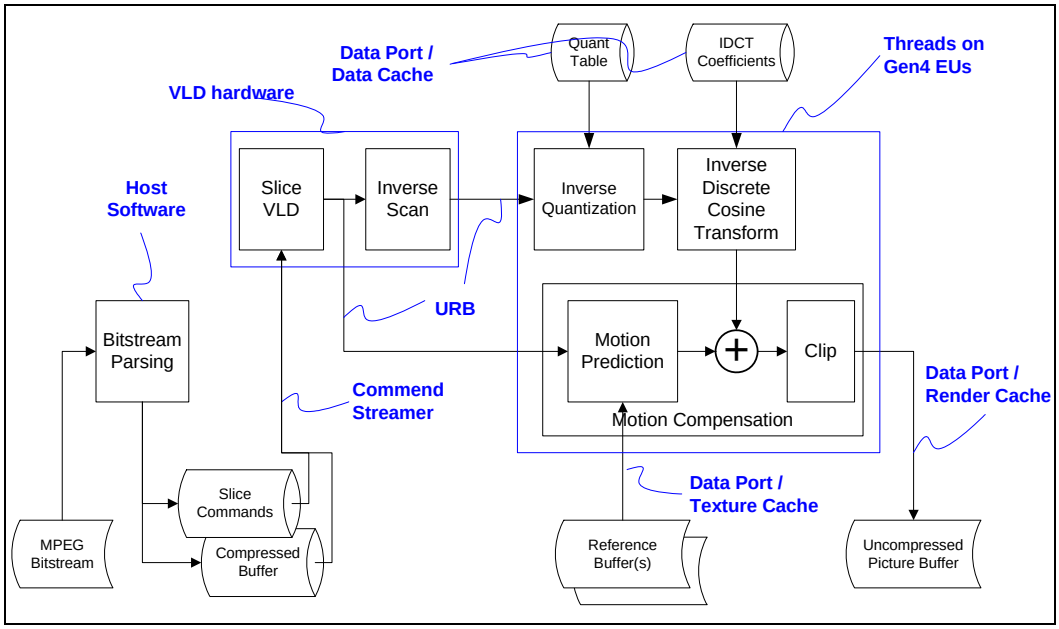


Table 10-14 shows the usage of GEN4 shared resource in this mode of operation.

Table 10-14. Use of GEN4 shared resources for post-VLD kernels

Shared Resources	Usage
VFE	VLD hardware to perform VLD and Inverse Scan.
TS	To dispatch root threads per macroblock.

Shared Resources	Usage
URB	For VFE to store macroblock data as payload of the root threads.
Data Port – Texture Cache	For reference data reads pointed by the motion vectors. Buffers support frame and field formats.
Data Port – Data Cache	For quantization matrix and DCT coefficient matrix reads.
Data Port – Render Cache	For render target writes. Buffers support frame and field formats.

10.9.1.2 Performance

This section covers the performance of the VLD fixed function and the estimated performance of the post-VLD kernel software.

The VLD hardware has a throughput of 1 symbol/clock but operates at half of the GEN4 execution core frequency. Assume the average symbol size is about 6 bits; it corresponds to a throughput of about 7 symbols/block for SD (standard resolution) MPEG-2 bitstream (MP@ML) at 10mbps rate or about 2.5 symbols/block for HD (high definition) MPEG-2 bitstream (MP@HL) at 20mbps. Therefore, the VLD hardware throughput is about 14 GEN4 clocks per SD block or 5 clocks per HD block.

Based on initial (rough) estimation of the post-VLD kernel code analysis, on a 16-EU configuration, it takes the EUs about 16 clocks to generate one block of data.

Therefore, for the 16-EU configuration, VLD hardware matches with the EU's performance for SD contents. There is sufficient performance headroom for HD contents. For configuration with less EUs, VLD hardware has sufficient performance headroom for both SD and HD contents.

10.10 Media Kernel Design Guide

As there is no fixed function specifically for other media functions, this section would outline the high level structure of support for each media feature. The detailed kernel descriptors will not be present in this document as they may be changed during driver/kernel co-development. The rest of the sections are kept brief.

10.10.1 MPEG-2 HWMC

Another commonly supported host-accelerator partition for MPEG-2 video decodes is Hardware Motion Compensation (HWMC) as shown in Figure 10-15. In this partition, the graphics accelerator only performs the last stage of the decode pipeline – motion compensation. This mapping can be summarized by the following points.

- Graphics driver assembles the per-macroblock HWMC kernel descriptors in memory. VFE unit takes the HWMC macroblock commands from the Command Streamer and stores the kernel descriptors into URB. Upon the completion of a macroblock, VFE unit signals TS unit that a new kernel is ready to be dispatched for execution.
- The per-macroblock HWMC kernel descriptors in URB are then forward by the TS unit to the GEN4 Thread Dispatcher to be dispatched as GEN4 threads on GEN4 Execution Units.

- The HWMC threads running on GEN4 EU performs MC. The IDCT block data (error data) is loaded by the thread from the Data Cache (or Texture Cache) via the Data Port using OWord Block Read message. The reference blocks in the reference pictures mapped to the Texture Cache are loaded from the Data Port using Media Block Read message. The final resulting macroblock data are written via the Data Port to the uncompressed picture buffer that is mapped to the Render Cache using Media Block Write message.

Figure 10-15. Functional mapping of MPEG-2 decode hardware acceleration with HWMC

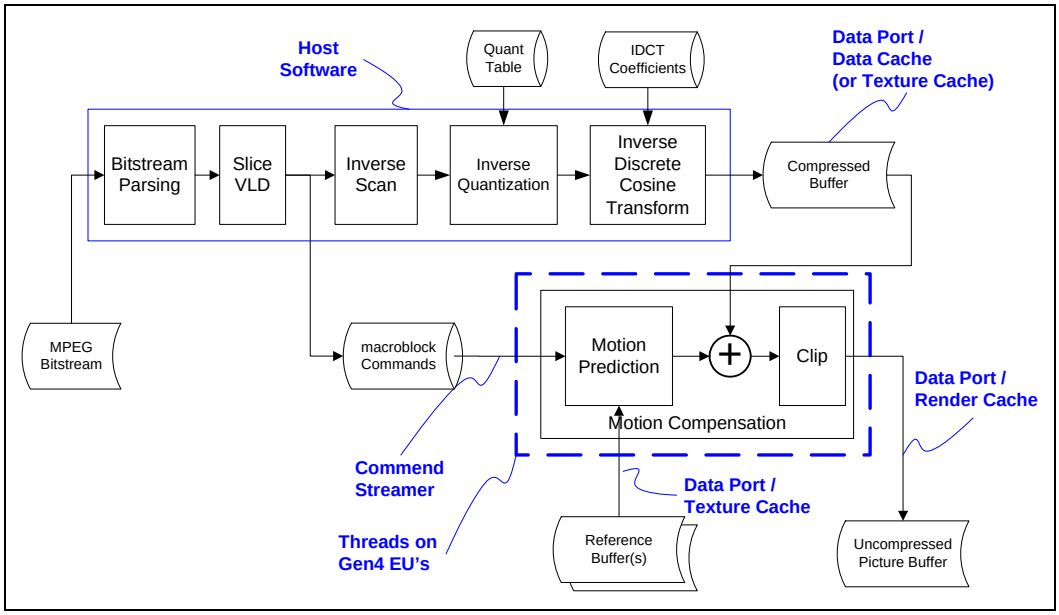


Table 10-15 shows the usage of GEN4 shared resource in this mode of operation.

Table 10-15. Use of GEN4 shared resources for HWMC kernels

Shared Resources	Usage
VFE	Simply forwarding macroblock kernel descriptor to TS.
TS	To dispatch root threads per macroblock.
URB	For VFE to store the payload of the root threads.
Data Port – Texture Cache	For reference data reads pointed by the motion vectors. Buffers support frame and field formats.
Data Port – Data Cache	For streaming of the correct data. (note: the correct data may also be mapped to the Texture Cache)
Data Port – Render Cache	For render target writes. Buffers support frame and field formats.

10.10.2 Deinterlace Filter

Computers commonly use a non-interlaced video display format, also called a progressive scan format, where an entire frame is scanned line-by-line. In contrast, many sources of consumer video such as NTSC/PAL television signals use an interlaced display format. Interlaced systems interleave two fields to display an entire frame. In order to display this interlaced material on a progressive scan computer display there is a need for deinterlacing.

A simple deinterlacing method is to apply a vertical filter to scale up a field to construct a frame for display. This method is called a line doubler in a progressive scan television set or called the 'Bob' method in PC graphics industry as different vertical offsets of the odd and even fields are adjusted as part of the up scaling. Due to the loss of vertical resolution within one interlaced field, flickering artifacts can be observed. This is more pronounced in relatively static areas that contain high vertical frequency details. Better deinterlacing methods are desirable.

Multi-frame pixel adaptive deinterlacing algorithms have been developed. As the implementations of these algorithms on GEN4 do not require specific hardware logic, descriptions of these algorithms are outside the scope of this Specification.

10.10.3 Video Encode

Various video encoding applications may also be realized on GEN4. Descriptions of these are outside the scope of this Specification.